VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

IMPLEMENTACE JEDNOTKY PRO VYHLEDÁVÁNÍ VZORŮ V FPGA

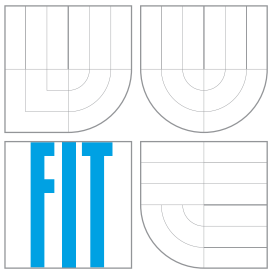
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. VLASTIMIL KOŠAŘ

BRNO 2010

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

IMPLEMENTACE JEDNOTKY PRO VYHLEDÁVÁNÍ VZORŮ V FPGA

IMPLEMENTATION OF THE PATTERN MATCHING UNIT IN THE FPGA

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. VLASTIMIL KOŠAŘ

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. JAN KAŠTIL

BRNO 2010

Abstrakt

Tato práce pojednává o algoritmech pro vyhledávání vzorů používaných v moderních systémech pro detekci nežádoucího provozu, přičemž se zaměřuje na algoritmy umožňující vyhledávání regulárních výrazů. Zabývá se přístupy založenými na deterministických a nedeterministických konečných automatech, hybridními přístupy a přístupem založeným na regulárních výrazech jako programovacím jazyku speciálních procesorů. Dále popisuje návrh implementace jednotek pro vyhledávání vzorů popsaných regulárními výrazy založenou na několika z popsaných přístupů včetně metodiky odhadu zabraných zdrojů. V další části je popsán vyvinutý softwarový systém pro generování jednotek. V následující části jsou ukázány a diskutovány dosažené výsledky.

Abstract

This master thesis focuses on algorithms for pattern matching used in modern IDS. The main focus is on regular expression matching. It deals with methods based on deterministic and nondeterministic finite automata, hybrid methods and with method based on regular expressions as programing langue for specialised processors. Implementation of pattern matching units based on some of descibed methodologies is described in next part. Methodology for resource consumption estimation is also described. Developed software system for unit generation is described in the next part. In the final part results are presented and discused.

Klíčová slova

Vyhledávání vzorů, regulární výrazy, konečné automaty, FPGA

Keywords

Pattern matching, regular expressions, finite automata, FPGA

Citace

Vlastimil Košař: Implementace jednotky pro vyhledávání vzorů v FPGA, diplomová práce, Brno, FIT VUT v Brně, 2010

Implementace jednotky pro vyhledávání vzorů v FPGA

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Jana Kaštila
Další informace mi poskytl Ing. Jan Kořenek
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Vlastimil Košař
26. května 2010

Poděkování

Chtěl bych poděkovat svému vedoucímu a všem, kteří mi poskytli odbornou pomoc.

Obsah

Seznam obrázků

Seznam tabulek

Kapitola 1

Úvod

V souvislosti s rozmachem internetu vzrostla potřeba monitorování sítě. Jedním z nástrojů pro monitorování sítě jsou systémy pro detekci nežádoucího provozu (IDS). Tyto systémy staví mimo jiné na vyhledávání vzorů, přičemž se stále více prosazují systémy vyhledávající vzory popsané regulárními výrazy namísto vzorů jenž jsou popsány pouze řetězci. Regulární výrazy umožňují vyšší flexibilitu a vyjadřovací sílu než pouhé řetězce.[20] Se vzrůstající rychlostí komunikačních linek roste potřeba akcelerace těchto systémů.

S nárůstem používání regulárních výrazů v IDS systémech vzrůstá také potřeba jejich efektivní implementace. Bylo vytvořeno mnoho různých algoritmů pro rychlé vyhledávání vzorů, jež se zaměřují na vysokou propustnost, co nejmenší množství spotřebovaných zdrojů, atd. Ukazuje se, že čistě softwarová řešení nestačí výkonově na vyhledávání regulárních výrazů při rychlostech linek větších než 1 Gb/s. Zde nastupují různé způsoby akcelerace vyhledávání vzorů, které si kladou za cíl vyhledávání i při rychlostech 10Gb/s a nastupujících 100Gb/s.

První kapitola se věnuje algoritmům pro vyhledávání vzorů v používaných v moderních IDS. Okrajově se věnuje algoritmům pro vyhledávání řetězců a převážně se věnuje algoritmům umožňujícím vyhledávání vzorů popsaných regulárními výrazy. Věnuje se základním přístupům jako jsou deterministické a nedeterministické konečné automaty. U nedeterministických konečných automatů se dále věnuje přístupům snižujícím velikost logiky zabrané na čipu. U deterministických konečných automatů představuje problém stavová exploze a přístupy řešící tento problém. Dále jsou představeny přístupy hybridní kombinující deterministické a nedeterministické konečné automaty. Na závěr je představen přístup pohlížející na regulární výrazy jako na programovací jazyk speciálních procesorů.

Druhá kapitola popisuje návrh jednotky umožňující vyhledávání vzorů popsaných regulárními výrazy, založenou na jednom z přístupů popsaných v předcházející kapitole. Dále obsahuje metodiku odhadu zabraných zdrojů pro tento návrh. Také se rozebírají vlastnosti a struktura vývojové platformy COMBOv2 a technologie Virtex 5 firmy Xilinx z hlediska vlastností využitelných při implementaci jednotky pro vyhledávání vzorů.

Třetí kapitola popisuje objektový návrh a implementaci rozšiřitelné knihovny pro generování jednotek pomocí různých algoritmů a třídy podpůrné. Knihovna je koncipována tak, aby byla snadno použitelná a rozšiřitelná.

Čtvrtá kapitola prezentuje a diskutuje dosažené výsledky pro implementované jednotky.

Závěr obsahuje shrnutí práce a diskutuje možnosti rozšíření.

Ze semestrálního projektu byla převzata kapitola první, která byla oproti semestrálnímu projektu doplněna, rozšířena a více ilustrována. Druhá kapitola byla také převzata ze semestrálního projektu a byla doplněna a rozšířena.

Kapitola 2

Algoritmy pro rychlé vyhledávání vzorů

Algoritmy pro rychlé vyhledávání vzorů můžeme rozdělit do dvou základních skupin:

- Do první skupiny patří algoritmy pro vyhledávání vzorů popsaných řetězci. Vyhledávání vzorů popsaných řetězci začíná být z moderních systémů pro detekci nežádoucího provozu (IDS) vytlačováno vyhledáváním vzorů popsaných regulárními výrazy. Regulární výrazy nabízí vyšší vyjadřovací sílu a flexibilitu než samotné řetězce [20]. Mezi algoritmy této skupiny patří například algoritmus Aho-Corasic, Boyer-Moore, Wu-Manber. Těmito algoritmy se již nebudu dále v této práci zabývat.
- Do druhé skupiny patří algoritmy pro vyhledávání vzorů popsaných regulárními výrazy. Vyhledávání pomocí vzorů popsaných regulárními výrazy nabízí větší flexibilitu a vyšší vyjadřovací sílu než vyhledávání pomocí vzorů popsaných řetězci. Do této skupiny patří algoritmy založené na konečných automatech a jejich modifikacích, jelikož podle teoretické informatiky jsou konečné automaty ekvivalentní regulárním výrazům.

Algoritmy pro vyhledávání podle vzorů popsaných regulárními výrazy můžeme podle způsobu jejich hardwarové implementace rozdělit na dva přístupy: logický a paměťový. Typickým představitelem logického přístupu je nedeterministický konečný automat (NFA). Typickým představitelem paměťového přístupu je deterministický konečný automat (DFA).[2]

Logický přístup spočívá v tom, že algoritmus je implementován převážně v logických obvodech a využívá tak implicitního paralelismu programovatelných hradlových polí (FPGA) a jejich snadné rekonfigurace. Paměťový přístup spočívá v tom, že algoritmus je implementován převážně v paměti. Tento způsob je vhodný například pro ASIC jež není možné rekonfigurovat pro změnu pravidel, nebo sériové stroje jako jsou například procesory.

Algoritmy pro vyhledávání vzorů budu dále nazývat také přístupy.

2.1 Regulární výrazy

Regulární výrazy můžeme definovat následovně [13]:

Definice Nechť Σ je konečná abeceda, pak regulární výrazy nad Σ a jazyky, které tyto regulární výrazy označují jsou definovány rekurzivně takto:

- $\emptyset$ je regulární výraz označující prázdnou množinu.

- ε je regulární výraz označující $\{\varepsilon\}$
- a je regulární výraz označující $\{a\}$, pro všechna $a \in \Sigma$
- Jsou-li p a q regulární výrazy označující jazyky P a Q, pak $(p + q)$, (pq) a (p^*) jsou regulární výrazy označující popořadě $P \cup Q$, $P.Q$ a P^*.
- Žádné jiné regulární výrazy, než regulární výrazy získané aplikací výše uvedených pravidel, nejsou regulární výrazy.

2.1.1 Regulární výraz neformálně

Regulární výraz (RV) je neformálně vzor vyhledávající jeden, nebo více řetězců. Jednotlivé znaky jsou regulární výrazy, jenž srovnávají samy sebe. Pokud r a s jsou regulární výrazy pak $r|s$, $r.s$, $r*$ a (r) jsou také regulární výrazy. $r|s$ vyhledává r nebo s, $r.s$ vyhledává r následované s, $r*$ vyhledává libovolný počet opakování r, operátory () jsou použity v běžném smyslu. Více například v [13].

V průběhu času byly standardní regulární výrazy rozšířeny o konstrukce zajišťující větší přehlednost zápisu (třídy znaků, počet opakování, ...) a zvyšující jejich sílu nad úroveň regulárních jazyků (zpětné reference,...). Mezi nejpoužívanější rozšíření patří např. PCRE (Perl Compatible Regular Expresion) [1].

2.1.2 PCRE

PCRE je rozšíření regulárních výrazů poprvé použité ve skriptovacím jazyce Perl, které se později rozšířilo do jiných jazyků a knihoven. Jedná se o jednu z praktických implementací regulárních výrazů, rozšiřující je jak z hlediska efektivnosti a přehlednosti zápisu, tak rozšiřující jejich výpočetní sílu, nad rámec klasických regulárních výrazů.

Příkladem rozšíření z hlediska efektivnosti a přehlednosti zápisu mohou být například znakové třídy, escape sekvence, přesné určení počtu opakování části regulárního výrazu, atd. Příkladem rozšíření zvyšující výpočetní sílu jsou zpětné reference.

Jelikož jsou konečné automaty a regulární výrazy ekvivalentní modely, použití rozšíření zvyšujících výpočetní sílu vede k nemožnosti vytvořit k danému PCRE ekvivalentní konečný automat.

Velmi dobrý popis PCRE je možné nalézt v článku [17].

2.2 Nedeterministický konečný automat

Do této skupiny patří přístupy založené na nedeterministickém konečném automatu a jeho modifikacích.

Definice Nedeterministický konečný automat M je pětice (Q, Σ, R, s, F)[13], kde:

- Q je konečná neprázdná množina stavů
- Σ je konečná vstupní abeceda
- $s \in Q$ je počáteční stav
- R je konečná množina pravidel tvaru $pa \to q$, kde $p, q \in Q, a \in \Sigma \cup \{\varepsilon\}$
- $F \subseteq Q$ je množina koncových stavů

2.2.1 Implementace

Implementace NFA jsou možné dvojího typu jako paměťové (např. pro sériové stroje - Procesor,...) nebo jako logické (FPGA), využívající implicitního paralelismu.

Výhody a nevýhody paměťového přístupu:

- Výhodou je menší paměťová náročnost než DFA $O(N)$.
- Nevýhodou je časová složitost pro jeden znak $O(N^2)$.[20]

Výhody a nevýhody logického přístupu:

- Časová složitost pro jeden znak $O(1)$ stejná jako u DFA, díky implicitnímu paralelismu.
- Nároky na velikost potřebné logiky a propojovací sítě odpovídají $O(N)$.[15]

Obecnou nevýhodou nedeterministických konečných automatů je to, že v nich může být zároveň aktivních až N stavů, což je problematické z hlediska vyhledávání výrazů v rámci celého síťového toku, protože pak je třeba uchovávat až pro každý tok až N stavů v registrech.

Konstrukce NFA z RV

Klasický přístup Konstrukce NFA z RV spočívá v konstrukci automatů pro jednotlivé znaky a jejich vzájemné spojování podle operátorů (| - sjednocení automatů, . - konkatenace automatů, * - iterace). Konstrukce se provádí na základě abstraktního syntaktického stromu daného regulárního výrazu. Více například v [13].

Přímá konstrukce z postfixové formy RV V článku [15] je ukázán algoritmus převodu regulárních výrazů v postfixové formě na odpovídající implementaci nedeterministického konečného automatu. Tento způsob konstrukce odstraňuje nutnost konstruovat z regulárního výrazu napřed nedeterministický konečný automat a následně z NFA konstruovat jeho FPGA implementaci. Nevýhodou je, že tento přístup je použitelný pouze tehdy, nepotřebujeme-li dělat žádné operace s NFA a optimalizace NFA, potřebné pro pokročilejší přístupy k implementaci NFA v FPGA.

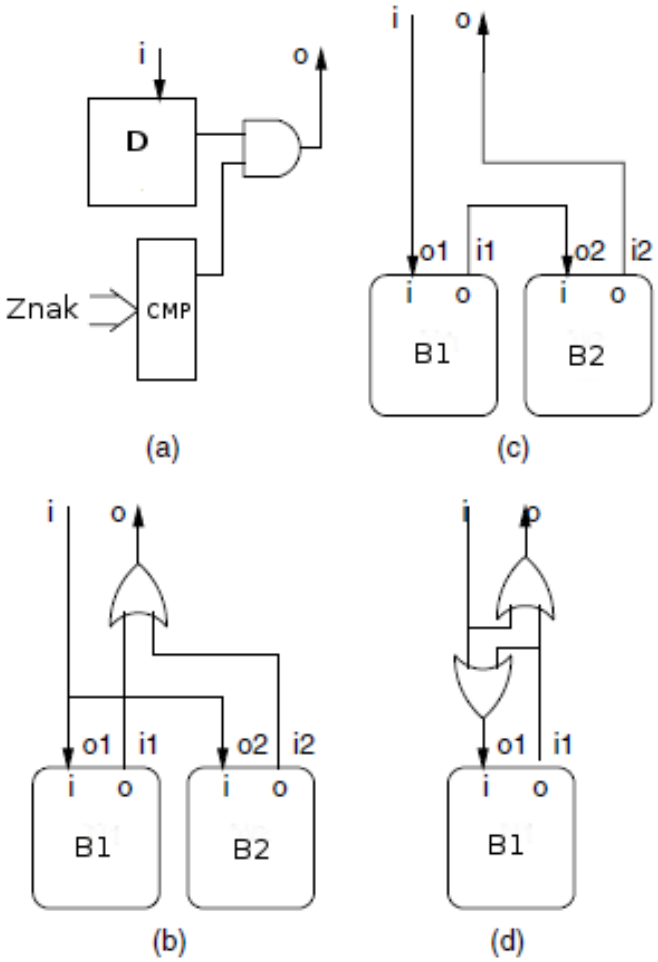
Postfixovou formu regulárního výrazu je možné získat postorder průchodem abstraktního syntaktického stromu daného regulárního výrazu. S pomocí zásobníku a funkcí pro umístění a propojení elementů (znak, *, |, .) je možné přímo generovat popis implementace v jazyce pro popis hardware (HDL).

Konstrukce HW implementace NFA

V článku [15] je popsán postup HW implementace NFA v FPGA. V následujících odstavcích bude shrnuta podstata tohoto přístupu.

Pro jednotlivé základní bloky a operátory RV lze vytvořit odpovídající bloky reprezentující je v FPGA. Jednotlivé stavy automatu jsou reprezentovány jako jednobitové registry. Blok jednoho znaku je tvořen registrem jehož výstup je hradlován výstupem komparátoru vstupního znaku (Efektivnější implementace obsahuje logiku před registrem.). Sjednocení (|) je implementováno jako dva bloky jejichž výstup je spojen logickým hradlem OR. Sjednocení (.) je implementováno prostým propojením výstupu prvního bloku na vstup druhého

bloku. Blok iterace (*) je implementován s pomocí dvou hradel typu OR na vstupu a na výstupu bloku. Schéma bloků je na obrázku 2.1. Kombinací těchto bloků je možno vytvořit strukturu odpovídající jakémukoliv danému regulárnímu výrazu.



Obrázek 2.1: Základní komponenty implementace NFA: a)základní blok b) r|s c) r.s d) r* (převzato [15], upraveno)

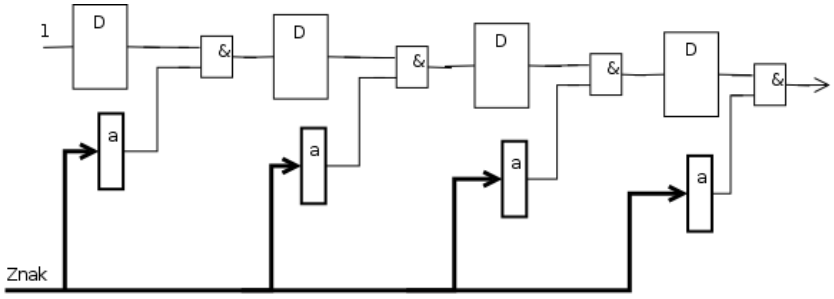
Pokud má stav pouze epsilon přechody na výstupu pak je možné odpovídající registr odstranit.

Tato implementace je schopná zpracovávat jeden znak každý takt hodin.

Na obrázku 2.2 je ukázána implementace regulárního výrazu $a\{4\}$. Tento regulární výraz byl zvolen tak, aby se na něm daly následovně ukázat další vylepšení HW implementace.

Vylepšení HW implementace NFA

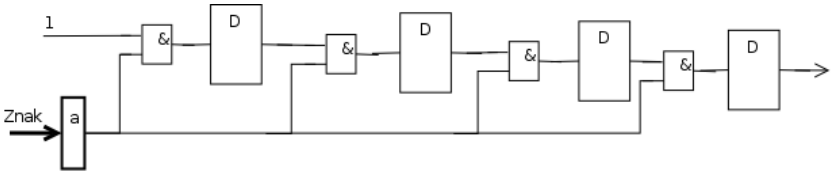
Sdílený dekodér znaků Clark aj. v [6] navrhl metodu snížení potřebné plochy na čipu u výše uvedeného řešení odstraněním dedikovaných komparátorů znaků pro každý stav automatu a jejich nahrazení sdíleným dekodérem znaků. Kromě úspory logiky na čipu dochází také k úspoře prvků propojovací sítě, protože je nyní ke každému registru stavu



Obrázek 2.2: Schéma implementace RV $a\{4\}$ základní metodou.

připojen pouze jednobitový signál namísto 8-bitového signálu.

Na obrázku 2.3 je ukázána implementace regulárního výrazu $a\{4\}$ s využitím sdíleného dekodéru znaků.

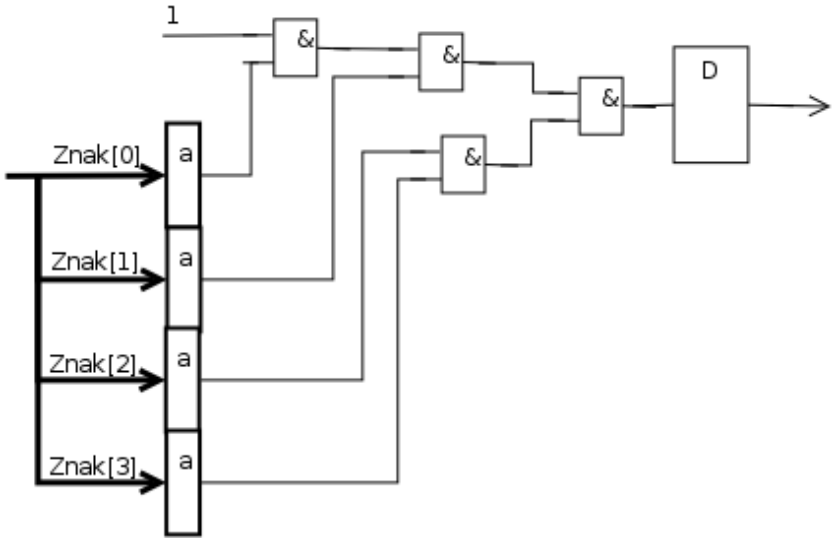


Obrázek 2.3: Schéma implementace RV $a\{4\}$ se sdíleným dekodérem znaků.

Rozšířený víceznakový automat (k-NFA) Klasický konečný automat přijímá nejvýše jeden znak každý takt hodin. Pro zvýšení počtu zpracovaných znaků je možné vytvořit rozšířený víceznakový automat přijímající více znaků zároveň. Základní myšlenkou rozšířeného víceznakového automatu je v rámci jednoho přechodu v rozšířeném víceznakovém automatu provést k přechodů původního nedeterministického automatu. Algoritmus pro převod automatu z NFA na k-NFA je popsán například v [3]. Jiný přístup pro generování automatu pro příjem více znaků přímo na implementační úrovni byl navržen v [19]. Výhodou tohoto přístupu je zvýšení rychlosti vyhledání vzoru n krát, čemuž odpovídá stejné zvýšení propustnosti řešení, ovšem za cenu zvýšení zabraných zdrojů a doby potřebné ke konstrukci takového automatu.

Na obrázku 2.4 je ukázána implementace regulárního výrazu $^{\wedge}a\{4\}$ s využitím rozšířeného víceznakového automatu (4 znaky).

Sdílení prefixů, infixů a sufixů Reálná použití regulárních výrazů mohou často obsahovat regulární výrazy jež sdílejí své prefixy, infixy, nebo sufixy. Sdílení prefixů je možné implementovat bez potřeby další HW jednotek. Provede se tak, že se prefix implementuje jako samostatný RV, jehož výstup je připojen na vstupy výrazů jichž je prefixem. [17] Sdílení infixů a sufixů je již složitější, protože si musíme pamatovat, pro který regulární výraz podvýraz pro infix/sufix provádíme. Toto lze například řešit pomocí uložení stavů a následného demultiplexoru jak je ukázáno v [12], místo demultiplexoru může být použito také hradlování hradlem AND, jak je ukázáno na obrázku 2.5. K zapamatování si údajů pro který z regulárních výrazů provádíme infix/sufix můžeme použít pokud známe přesnou



Obrázek 2.4: Schéma implementace RV ^$a\{4\}$ pomocí rozšířeného víceznakového (4 znaky) automatu.

délku doby provádění posuvný registr (snadná implementace pomocí elementů FPGA) nebo pokud nemůžeme dobu určit pak musíme tuto informaci připojit ke každému registru stavu (vyskytuje-li se v infixu/sufixu konstrukce typu .* a podobná opakování).

Na obrázku 2.5 je ukázán způsob implementace prefixů, infixů a sufixů na blokové úrovni. Bloky paměti vstupu mohou být implementovány jedním ze dvou výše uvedených přístupů v závislosti na tom jaké konstrukce RV infix/sufix obsahuje.

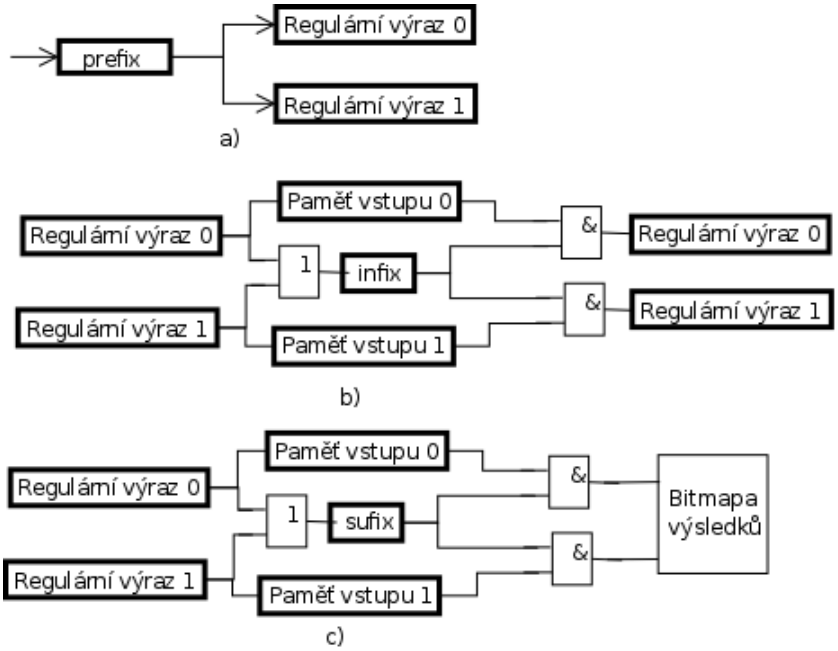
Z pohledu výpočetní náročnosti je hledání infixů nejnáročnější operací, protože infixy se mohou v automatech vyskytovat na kterémkoliv místě narozdíl od prefixů a sufixů, které se vyskytují na začátku a nebo na konci.

Nové bloky pro podporu PCRE Pro snížení počtu zabraných zdrojů byly v [17] navrženy nové bloky pro podporu PCRE. Jedná se zejména o konstrukce typu m, n, určující počet opakování nějakého bloku. Jsou implementovány s pomocí posuvných registrů a čítačů. Dále ukazují možnost jak transformovat typickou implementační konstrukci skládající se z hradla typu or a registru na registr na jehož resetovací vstup je přivedena negovaná hodnota výskytu daného znaku přechodu. Dále zavádí sdílení znakových tříd v blocích generování znakových tříd místo generování dedikovaných znakových tříd pro každý jejich výskyt.

V následujících odstavcích budou představeny jednotlivé bloky a vylepšení implementace.

Bloky opakování podporují pouze opakování znaků nebo třídy znaků, pro opakování části regulárních výrazů je třeba provést jejich rozbalení. Autoři zjistili, že implementace těchto bloků má smysl, protože 95% opakování v pravidlech Snort v2.4 je tvořeno opakováním znaku nebo třídy znaků.

Blok přesného počtu opakování Tento blok slouží k detekci opakování přesně N stejných znaků a. Blok $a\{N\}$ implementuje konkatenaci N znaků a. Tudíž je možné tento



Obrázek 2.5: Blokové schéma implementace a) prefixů, b) infixů a c) sufixů

blok definovat jako:

$$a\{N\} = \begin{cases} \varepsilon & \text{pro } N = 0 \\ a & \text{pro } N = 1 \\ a \ldots a, \text{ n-krát} & \text{pro } N > 0 \end{cases} \tag{2.1}$$

Tento blok je možné efektivně implementovat v technologii FPGA Virtex firmy Xilinx pomocí primitiva FPGA SRL16 implementující posuvný registr, čítače a několika registrů. SRL16 implementuje 1-bitový posuvný registr o 16 položkách a zabírá 1 LUT. Schéma obvodu je na obrázku 2.6 převzatém z [17].

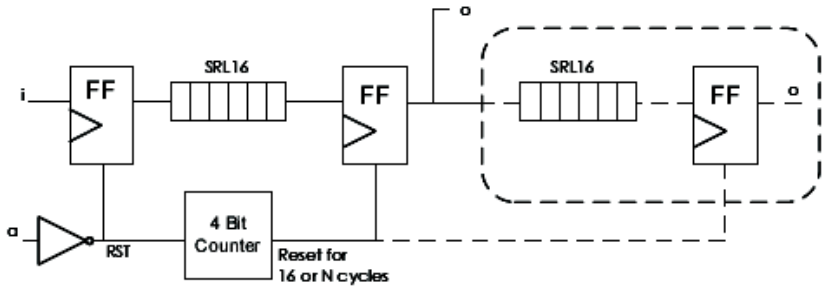
Tento blok funguje následujícím způsobem. Je-li přijat token i, pak je zpropagován na výstup po N úspěšných srovnáních. Token i vstupuje do posuvného registru došlo-li k úspěšnému porovnání znaku a, jinak je registr resetován. Registr je posunován po dobu N taktů hodin nedošlo-li k neúspěšnému porovnání se znakem a. V případě neúspěšného porovnání je nutné provést reset posuvného registru. Jelikož je element SRL16 neresetovatelný, je mezi jeho jednotlivé instance vložen klopný obvod sloužící k resetování obsahu v součinnosti se 4-bitovým čitačem zajišťujícím dobu resetování 16 taktů hodin.

Prostorová složitost řešení je $O(N)$. Autoři uvádějí, že blok pro RV $a\{1000\}$ bylo spotřebováno 63 konfigurovatelných logických bloků. Jeden element SRL16 a jeden klopný obvod je namapovatelný do jednoho slice.

Blok opakování alespoň N Tento blok slouží k detekci opakování alespoň N stejných znaků. Blok $a\{N,\}$ implementuje konkatenaci alespoň N znaků a. Tudíž je možné tento blok definovat jako:

$$a\{N,\} = \cup_{i=N}^{\infty} a\{i\} \tag{2.2}$$

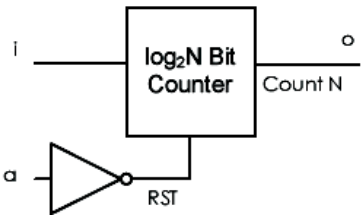
Autoři v článku [17] dokázali, že výstup tohoto bloku závisí pouze na prvním vstupním tokenu i po posledním restartu a nezávisí tudíž na následujících tokenech. Díky tomu je



Obrázek 2.6: Schéma obvodu implementujícího blok opakování právě N

možné tento blok implementovat jako čítač řízený prvním tokenem přijatým po restartu. Čítač čítá do N a zůstává v hodnotě N se zapnutým výstupem dokud nedojde k neúspěšnému porovnání se znakem a.

Schéma obvodu je na obrázku 2.7 převzatém z [17].



Obrázek 2.7: Schéma obvodu implementujícího blok opakování alespoň N

Prostorová složitost řešení je $O(log_2(N))$.

Blok opakování M až N Tento blok slouží k detekci opakování M až N stejných znaků. Blok $a\{M, N\}$ implementuje konkatenaci nejméně M a nejvíce N znaků a. Tudíž je možné tento blok definovat jako:

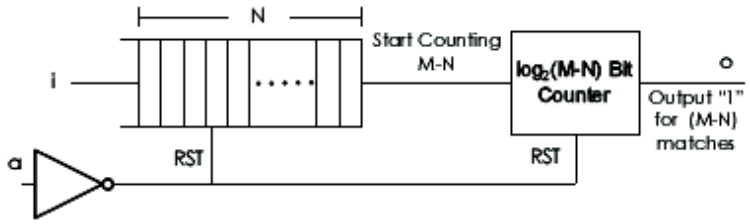
$$a\{M, N\} = \cup_{i=M}^{N} a\{i\} \tag{2.3}$$

Tento blok je možné sestavit jako blok opakování 0 až N a z bloku přesného počtu opakování. Blok opakování 0 až N je možné sestavit z jednoho čítače, který začíná čítat od nuly do N a pak se restartuje do nuly. K restartu dojde také pokud dojde k neúspěšnému porovnání se znakem a. V článku [17] autoři dokázali, že výstup tohoto bloku závisí pouze na posledním tokenu přijatém po posledním neúspěšném porovnání se znakem a, tudíž je výše uvedený přístup k implementaci správný. Blok opakování je pak sestaven z bloku přesného opakování na jehož výstup je napojen blok opakování 0 až N.

Vstupní tokeny i vstupují do posuvného registru délky N, který je resetovatelný při neúspěšnému porovnání se znakem a. (Implementace tohoto chování posuvného registru viz výše.) Po N úspěšných srovnáních bude na výstupu posuvného registru 1 a tudíž dojde k aktivaci čítače čítajícího do $N - M$.

Schéma obvodu je na obrázku 2.8 převzatém z [17].

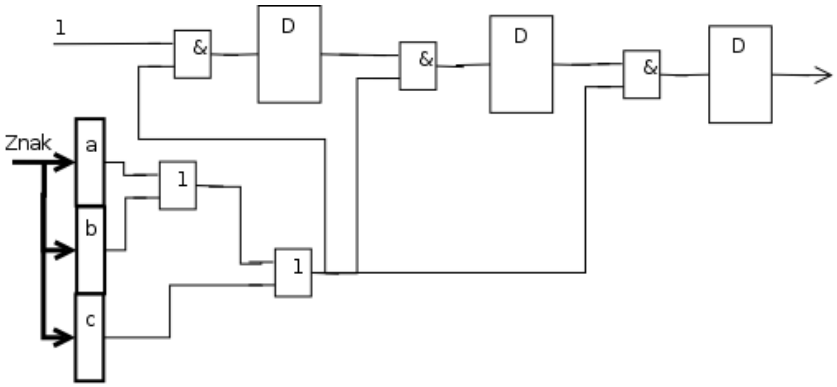
Prostorová složitost řešení je $O(N + log_2(N - M))$. Díky použití logických elementů FPGA SRL16 je velikost implementace relativně malá.



Obrázek 2.8: Schéma obvodu implementujícího blok opakování M až N

Sdílení tříd znaků Třídy znaků jsou hojně používány v pravidlech systému Snort. Každá třída znaků je sjednocením několika znaků. Samostatnou implementací tříd znaků a sdílením jejich výstupů je možné dosáhnout snížení velikosti zabrané plochy čipu. Použití sdílení tříd znaků redukuje podle autorů článku 8000 znakových tříd na 62 unikátních případů při použití RV programu Snort.

Schéma obvodu implementující sdílení třídy znaků je na obrázku 2.9.



Obrázek 2.9: Schéma obvodu implementujícího sdílení třídy znaků pro regulární výraz $[abc][abc][abc]$

Sdílení statických vzorů Při reálném použití regulárních výrazů (např. programu Snort) dochází k tomu, že vzniká velké množství statických podvzorů (řetězce), které můžou být vyhledávány samostatně s pomocí některého efektivního přístupu pro hledání vzorů popsaných řetězci. Autoři článku navrhují použití jimi navrženého přístupu s DCAM se sdílením výstupů. Také se využívá sdíleného ASCII dekodéru a sdílení posuvných registrů. Autoři uvádějí, že regulární výrazy programu Snort v2.4 obsahují 2000 jedinečných statických podvzorů čítajících dohromady cca. 35000 znaků.

2.3 Deterministický konečný automat

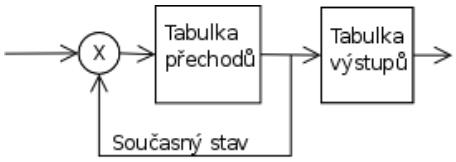
Do této skupiny patří přístupy založené na deterministickém konečném automatu a jeho modifikacích.

Definice Deterministický konečný automat M je pětice (Q, Σ, R, s, F)[13], kde:

- Q je konečná neprázdná množina stavů
- Σ je konečná vstupní abeceda
- $s \in Q$ je počáteční stav
- R je konečná množina pravidel tvaru $pa \rightarrow q$, kde $p, q \in Q, a \in \Sigma$, přičemž pro každé $pa \rightarrow q \in R$ platí, že množina $R \setminus \{pa \rightarrow q\}$ neobsahuje žádné pravidlo s levou stranou pa
- $F \subseteq Q$ je množina koncových stavů

2.3.1 Implementace

Na obrázku 2.10 je vidět základní schéma implementace konečného automatu typu DFA.



Obrázek 2.10: Schéma implementace DFA

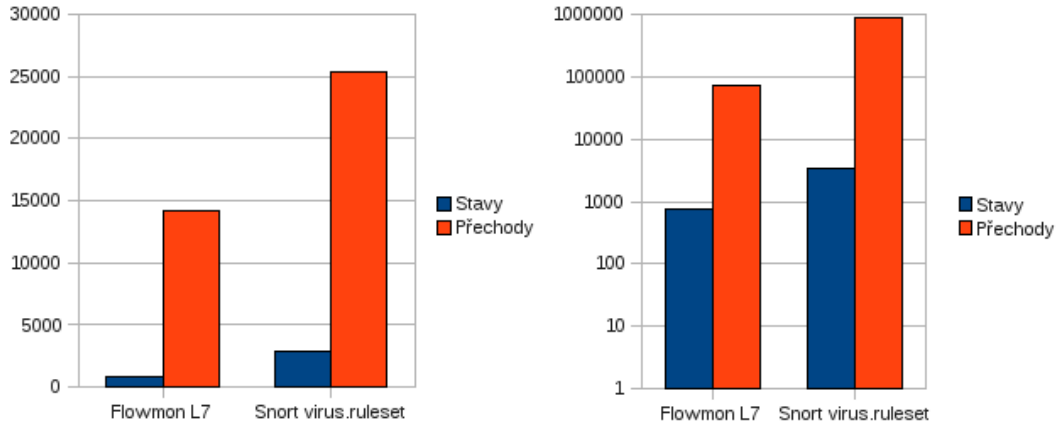
Výhody a nevýhody DFA:

- Výhodou je jeho časová složitost pro jeden znak $O(1)$.
- Výhodou je obvykle paměťová varianta implementace umožňující snadnou změnu RV.
- Výhodou je snadná použitelnost pro vyhledávání regulárních výrazů v rámci toků, protože je třeba uchovávat informaci pouze o jednom aktuálním stavu.
- Nevýhodou je jeho paměťová náročnost $O(2^N)$.
- Nevýhodou je také náročnost převodu NFA na DFA $O(2^N)$.

2.3.2 Stavová exploze

Jedním z největších problémů deterministických konečných automatů je teoretický maximální možný počet stavů a přechodů DFA jenž činí 2^N, kde N je počet stavů NFA. Tento počet sice nebývá v praxi dosažen, ale i tak bývá nárůst značný. [2] Příčin stavové exploze je několik. Příčiny je možné rozdělit do dvou skupin. První skupina jsou příčiny v regulárních výrazech. Jsou to některé operátory regulárních výrazů (+,*), případně operátory rozšířených regulárních výrazů (PCRE - xm,n značící m až n opakování). Tyto operátory jsou v praktických regulárních výrazech časté, jsou primárně používány pro detekci podvzorů oddělených nějakým počtem znaků. Obecně lze říci, že podautomaty budou replikovány ve výsledném automatu jednou pro každý výskyt .* .[2] Druhou příčinou stavové exploze je sdružování jednotlivých regulárních výrazů do jednoho velkého konečného automatu.[2]

Na grafu 2.11 je znázorněn počet stavů a přechodů NFA a DFA (pravý graf má osu Y v logaritmickém měřítku) pro dva různé soubory regulárních výrazů.



Obrázek 2.11: Počet stavů a přechodů v NFA (vlevo) a v DFA (vpravo) pro dva různé soubory RV (pravý graf má osu Y v logaritmickém měřítku)

2.3.3 Přístupy řešení stavová exploze

mDFA

Jedním z řešení stavové exploze je nesdružovat jednotlivé regulární výrazy do jednoho velkého DFA, ale sdružovat je do několika menších DFA, čímž dojde k tomu že efekty stavové exploze budou menší a celková velikost potřebné paměti bude menší. Tyto menší DFA mohou pracovat paralelně. V závislosti na implementaci může dojít ke zvýšení nároků na propustnost paměti (Při architektuře s jednou sdílenou pamětí). [14] [20]

D^2FA

Delayed Input DFA (D^2FA) byl zaveden v [8]. Jedná se o rozšíření standardního DFA o takzvané implicitní přechody, které se provedou pokud není pro daný znak na vstupu možno provést přechod. Každý stav může mít nejvýše jeden implicitní přechod. Implicitních přechodů může být provedeno pro jeden vstupní znak několik. Tato úprava DFA je založena na zjištění, že mnoho stavů má společné množství přechodů a vytváří se tak redundance. Převod DFA na D^2FA je NP-těžký problém, tudíž bylo pro efektivní převod nutno nalézt heuristiku. Nevýhodou tohoto přístupu je to, že každé provedení implicitního přechodu způsobuje prodloužení doby zpracování jednoho znaku oproti DFA. Výhodou je značná redukce potřebného paměťového prostoru (podle autorů až o 95%).

CD^2FA

Content Addressed Delayed Input DFA (CD^2FA) byly zavedeny v [9] a jsou modifikací D^2FA. Místo adresace stavu číslem jak je běžné u DFA, CD^2FA provádí adresaci stavu popisky obsahu (content labels), které obsahují část informací bývajících obvykle v tabulce přechodů. Každý stav má svůj popisek. Popisek se skládá z množiny znaků pro daný stav, všech jeho předchůdců ve stromu implicitních přechodů a z identifikátoru stavu v kořeni tohoto stromu. Pro efektivní adresaci stavů se používá hashovacích funkcí. Vstupem algoritmu pro vytvoření CD^2FA je D^2FA. Pro dosažení nejlepších výsledků je potřeba provádět některé optimalizace(redukce CD^2FA, redukce abecedy, ...). CD^2FA dosahují přibližně stejné

redukce potřebného paměťového prostoru jako D^2FA, ale dosahují vyšší propustnosti než D^2FA (srovnatelné s klasickými DFA).

XFA

Model XFA (Extended Finite Automata) zavedený v [16] rozšiřuje DFA o konečnou množinu pomocných proměnných a o explicitní instrukce pro změnu jejich stavu připojené ke stavům XFA. Pomocné proměnné nemohou ovlivnit přechod, avšak mohou ovlivnit chování koncového stavu a tím zredukovat potřebné množství stavů a přechodů. Pro efektivní sjednocování XFA je potřeba provádět optimalizace instrukcí a počtu proměnných pro efektivní vyhodnocení. Použití XFA redukuje velikost potřebné paměti, ale potřebuje pro každý stav spouštět odpovídající kód což má dopad na propustnost.

Hierarchické DFA

Hierarchický přístup spočívá v tom, že jsou nalezeny segmentované podvýrazy daného RV v jednom DFA a souběžně běžící druhý DFA identifikuje korelaci segmentovaných podvýrazů a tím provádí vlastní vyhledávání. Segmentace se provádí tak, že například RV $R.*S.*T$ je segmentován na podvýrazy R, S a T, tj. podvýrazy jsou části RV oddělené konstrukcemi typu $.*$. Tento postup podle autorů dovoluje dosáhnout redukce i více než 90%. Při konstrukci hierarchického DFA je třeba dodržet jednu podmínku, jinak může dojít k generování false-positiv výsledků (více v článku [11]).

2.3.4 Jiné přístupy optimalizace

Znakové třídy pro redukci velikosti tabulky přechodů

V článku [5] byl navržen přístup komprese tabulky přechodů zavedením tříd znaků. Zavedení tříd znaků znamená to, že je určitá skupina znaků splňující určité kritérium pomocí předřadného dekodéru sloučena v jednu třídu znaků. Podmínkou pro to, aby mohla být určitá skupina znaků sjednocena v třídu znaků je, že pro všechny stavy automatu musí platit právě jedna ze dvou následujících podmínek:

- Existují přechody pro všechny symboly ve skupině takové, že jejich cílový stav je pokaždé stejný.
- Neexistuje žádný přechod pro všechny symboly ve skupině.

Po vytvoření třídy znaků je zvoleno její číslo. Může to být například číslo jednoho znaku, nebo může být dle potřeby vytvořen kód úplně jiný.

Na obrázku 2.12 je ukázána přechodová tabulka před a po vytvoření tříd znaků. Vlevo je přechodová tabulka bez tříd znaků, vpravo je tabulka přechodů po vytvoření tříd znaků. Byly vytvořeny dvě třídy znaků X a Y. Třída znaků X je složena ze znaků a a b a třída znaků Y je složena ze znaků a a b. Znak e není možné do třídy Y zahrnout, protože by nebyla splněna jedna z výše uvedených podmínek.

Úspěšnost této metody závisí na podobě regulárních výrazů a také na počtu stavů. V nepříznivém případě se nemusí podařit nalézt žádná třída znaků, protože pro zadanou množinu regulárních výrazů nebude možné najít žádnou takovou množinu znaků, která splní výše uvedenou podmínku.

Implementace se provádí zavedením dekodéru znaků, který provede dekódování znaků z kódu ASCII na kód odpovídající dané třídě znaků pokud je použita. Pokud není použita,

Stav \ Symbol	a	b	c	d	e
0					1
1			2	2	2
2	3	3	2	2	2
3	0	0			

Stav \ Symbol	X	Y	e
0			1
1		2	2
2	3	2	2
3	0		

Obrázek 2.12: Komprese přechodové tabulky DFA pomocí tříd znaků

pak se dekódování neprovede a je vrácena původní hodnota. V FPGA se jako ideální jeví implementace pomocí pamětí BRAM, kde původní kód znaku představuje adresu do paměti BRAM a v adresované buňce je uložena nová hodnota. V případě použití dvouportové paměti BRAM je možné provádět najednou dva překlady z jednoho kódu do druhého.

Tento přístup k optimalizaci je možné použít i jako předřadný dekodér pro přístupy založené na nedeterministickém konečném automatu. V tom případě se projeví, při jeho použití, snížení množství LUT spotřebovaných sdílenými třídami znaků, protože se část znakových tříd přesune do předřadného dekodéru.

2.4 Hybridní přístup

Hybridní přístupy kombinují deterministické a nedeterministické konečné automaty.

2.4.1 Hybridní DFA přístup

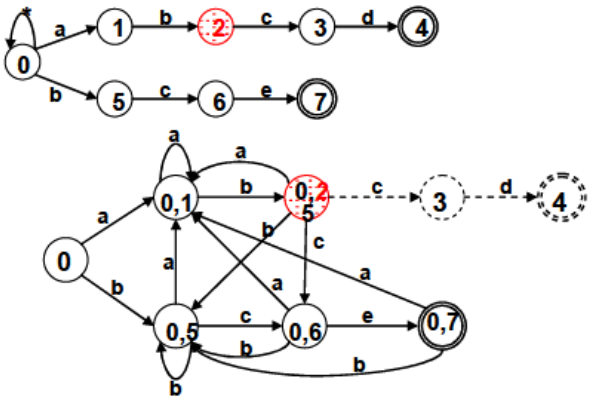
Michela Becchi a Patrick Crowley navrhli v [2] přístup kombinující použití deterministických a nedeterministických konečných automatů za účelem vyřešení stavové exploze k níž dochází při determinizaci nedeterministických konečných automatů. Při konstrukci hybridního konečného automatu (HFA), všechny uzly přispívající ke stavové explozi budou uloženy jako NFA a všechny ostatní budou uloženy jako DFA. V rámci jednoho regulárního výrazu nejvíce přispívají ke stavové explozi konstrukce typu $.*$ (v případě rozšířených regulárních výrazů - PCRE, ... - také konstrukce typu $x\{m,n\}$). Tyto konstrukce jsou relativně snadno detekovatelné.

Hybridní konečný automat se skládá ze stavů typu DFA, stavů typu NFA a z hraničních stavů. Příklad HFA je na obrázku 2.13. Konstrukce podmnožiny stavů jdoucích do DFA části začíná od počátečního stavu nedeterministického konečného automatu a pokračuje dokud není dosaženo podmínky pro přechod z DFA části do NFA části.

Výsledný hybridní konečný automat bude mít následující příznivé vlastnosti (z hlediska sériové implementace - př. procesor):

- Počáteční stav bude stavem DFA.
- NFA část automatu bude aktivována až při dosažení hraničního stavu.
- DFA část není zpětně aktivována NFA částí.

V tomto navrženém přístupu se počítá s paměťovou implementací NFA částí. Z čehož plyne potenciálně vysoký nárůst potřebné paměťové propustnosti, protože NFA automat může mít teoreticky všechny stavy aktivní. Za účelem vyhnutí se tomuto případu je možné



Obrázek 2.13: Hybridní konečný automat, tečkovaně NFA, červeně hraniční stav (převzato z [2])

převést koncové NFA automaty na DFA automaty a tím ušetřit propustnost za cenu možného zvýšení počtu stavů.

2.4.2 Hledání deterministických částí

Tento algoritmus je vyvíjen na Ústavu počítačových systémů Fakulty informačních technologií Vysokého učení technického v Brně. Informace o tomto algoritmu je možné najít v článku [7]. Přístup je založen na zjištění, že v NFA ve většině případů nejsou během vyhledávání aktivní všechny stavy najednou. Na obrázku 5.2 je experimentálně zjištěný histogram počtu naráz aktivních stavů NFA pro množinu RV backdoor. Najdeme-li takovou množinu stavů, které nebudou nikdy aktivní zároveň, pak je možné tyto stavy implementovat v DFA.

Pro nalezení této množiny stavů (bezkolizní množiny) je třeba nejdříve provést převod NFA na DFA a při převodu je třeba si pro každý stav DFA pamatovat z jakých stavů NFA vznikly. Poté je třeba pro každý stav NFA nalézt všechny stavy s ním kolizní tak, že pokud je stav NFA s nějakými dalšími stavy NFA v nějakém stavu DFA, pak je s těmito stavy v kolizi.

Na základě množin kolizních stavů je možné vypočítat bezkolizní množinu. Bezkolizních množin může být více, proto se pokoušíme najít největší bezkolizní množinu. To je však výpočetně náročné, proto je potřeba použít heuristiky. Heuristik je několik, základní je ta, že pokud stav nemá žádné kolize pak je automaticky součástí bezkolizní množiny. Myšlenka další heuristiky spočívá v tom, že se vybírají prvky s nejméně kolizemi. Pokud existuje těchto stavů více rozhodneme se podle toho, kolik kolizí bude odstraněno použijeme-li tento stav. Aplikací těchto heuristik je možné zkonstruovat bezkolizní množiny. Bezkolizních množin můžeme v jednom konečném automatu nalézt více. Nalezneme-li největší bezkolizní množinu, pak můžeme buď zbytek stavů prohlásit za množinu stavů v kolizi a skončit, nebo můžeme pro tuto zbylou množinu stavů opakovaně používat heuristiky a provádět výpočet dalších bezkolizních množin.

Stavy bezkolizní množiny tedy mohou být implementovány v DFA, zatímco zbytek stavů zůstává v NFA. Bylo-li nalezeno více bezkolizních množin, pak můžeme každou z nich implementovat v samostatném DFA.

Stavy části konečného automatu (deterministické i nedeterministické) do kterých vede přechod z jiné části konečného automatu se nazývají vstupní stavy. Stavy nějaké části

konečného automatu (deterministické i nedeterministické) do kterých vede přechod z dané části konečného automatu se nazývají výstupní stavy.

Na přechodu do deterministické části automatu se nachází vstupní dekodér provádějící dekódování vstupních stavů z kódu 1 z n na binární kód. Na výstupu deterministické části automatu se nachází dekodér z binárního kódu na kód 1 z n.

Při implementaci přechodové tabulky deterministické částí v FPGA je třeba mít na zřeteli, že přechodová tabulka je typicky řídce zaplněna a pro použití v FPGA je nutné využít vhodný způsob implementace. Takovým způsobem je překrývání řádků tabulky. V tomto případě je přechodová tabulka implementována jako jednorozměrné pole, do něhož jsou řádky tabulky přechodů namapované takovým způsobem, že se jednotlivé řádky mohou překrývat. Adresace pole se provádí na základě výpočtu adresy ze stavu a vstupního symbolu. Pro výpočet adresy je možné použít různé metody, ale z hlediska rychlosti se jako vhodné jeví použití funkce XOR. Daní za tento způsob adresace je to, že může docházet ke kolizím. Problém kolizí je řešen tak, že je v poli uložen kromě následujícího stavu i symbol, se kterým se má přechod provést. Pro nalezení ideálního mapování (uspořádání) by bylo třeba projít $\begin{pmatrix} m \\ n \end{pmatrix}$ kombinací, kde m je počet řádků tabulky přechodů a n je velikost paměti, a tudíž je třeba použít neoptimální, ale výpočetně jednodušší způsob výpočtu mapování. Mapování se provádí tak, že je hledáno vhodné očíslování stavů tak, aby potřebná velikost paměti pro uložení pole byla co nejmenší, V následujících dvou podkapitolách jsou ukázány dva přístupy jak toto mapování řešit. V tabulce 5.3 je ukázáno, že oba dva přístupu jsou si do určité míry rovnocenné.

Heuristika

V článku [7] je představen způsob, jak vyřešit uspořádání řádků tabulky přechodů pomocí heuristiky. Základní myšlenkou této heuristiky je to, že je třeba seřadit řádky tabulky přechodů podle míry zaplnění řádku od největšího po nejmenší a při umísťování řádků do paměti postupovat od nejzaplněnějšího po nejméně zaplněné řádky tabulky přechodů. Tento postup zajistí to, že řádky s malým zaplněním budou moci být umísťovány do mezer vzniklých při umisťování řádků s velkým zaplněním.

Genetický algoritmus

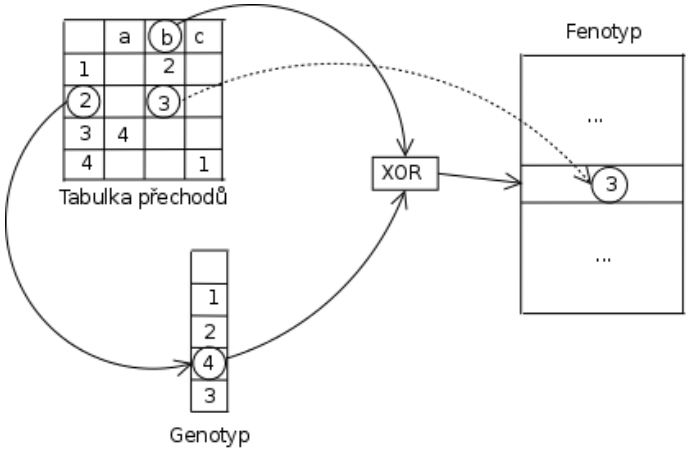
Jako alternativu k heuristice jsem navrhl použití genetického algoritmu pro nalezení uspořádání řádků přechodové tabulky. Pro použití této metody výpočtu mapování je nutné nastavit maximální velikost paměti, kterou chceme použít.

Genotypem je pole čísel daného rozsahu. Rozsah je dán maximální požadovanou velikostí obsazené paměti překrytou tabulkou přechodů. Pole čísel v genotypu představuje mapování očíslování stavu konečného algoritmu na výsledné očíslování, které se použije při překrývání řádků. Fenotypem je výsledné uspořádání řádků tabulky přechodů v paměti. Uspořádání se vypočítá tak, že výsledná adresa v paměti se pro každou dvojici (stav, symbol), pro kterou existuje v tabulce přechodů přechod vypočítá následovně:

$$adresa = genotyp[stav] XOR symbol \tag{2.4}$$

Na takto vypočítané místo se uloží do paměti (fenotypu) potřebné informace. Schéma výpočtu uspořádání je na obrázku 2.14.

Evaluace probíhá tím způsobem, že je z genotypu vytvořen fenotyp a pro každý element výsledné paměti je spočítáno kolikrát je použit. Výsledná fitness funkce se vypočítá tak, že



Obrázek 2.14: Převod genotypu na fenotyp.

se sečte počet použití zmenšený o jedničku u elementů použitých více než jedenkrát. Mají-li všechny elementy počet použití roven nule nebo jedna, pak se evoluce podařila a byl nalezen výsledek (Hodnota fitness funkce je pak 0). Evoluční algoritmus se tedy snaží snižovat hodnotu fitness funkce. Je-li hodnota fitness rovná jedné, pak se začíná používat varianta fitness funkce, která ohodnocuje kvalitu nalezeného řešení a provádí tudíž optimalizaci na velikost paměti. Řešení je tím kvalitnější, čím je velikost zabrané paměti menší. Od výsledku normální fitness funkcce se v tomto případě odečítá rozdíl maximální velikosti paměti a použité velikosti paměti. Fitness funkce tudíž dále klesá do záporných čísel.

Genetický algoritmus používá při tvorbě nové generace operaci jednobodového křížení a mutaci. Selekce kandidátů pro křížení probíhá pomocí výběru rankem. Zkřížený jedinec může být následně zmutován. Pokud se má použít pouze mutace, pak dochází k mutaci pouze nejlepšího jedince. Používán je také elitismus a tudíž nejlepší jedinec stávající generace se vyskytuje i v generaci následující.

2.5 Procesorový přístup

Ivano Bonesana, Marco Paolieri a Marco D. Santambrogio navrhli v [4] nový přístup k problému vyhledávání vzorů. Pohlížejí na regulární výrazy jako na programovací jazyk pro dedikované procesory a tudíž nepotřebují konstruovat deterministické nebo nedeterministické konečné automaty. Navržená architektura procesoru je schopná načítat regulární výrazy uložené v instrukční paměti a provádět vyhledávání v textovém řetězci uloženém v paměti. Návrh procesoru a překladače byl inspirován architekturou procesorů VLIW, což umožňuje vyvážit parametry procesoru (výkon, spotřeba, ...). Procesor průměrně dosahuje více než jednoho zpracovaného znaku za takt a potřebné množství paměti je lineární. Procesorový přístup umožňuje snadnou modifikaci vyhledávaných regulárních výrazů.

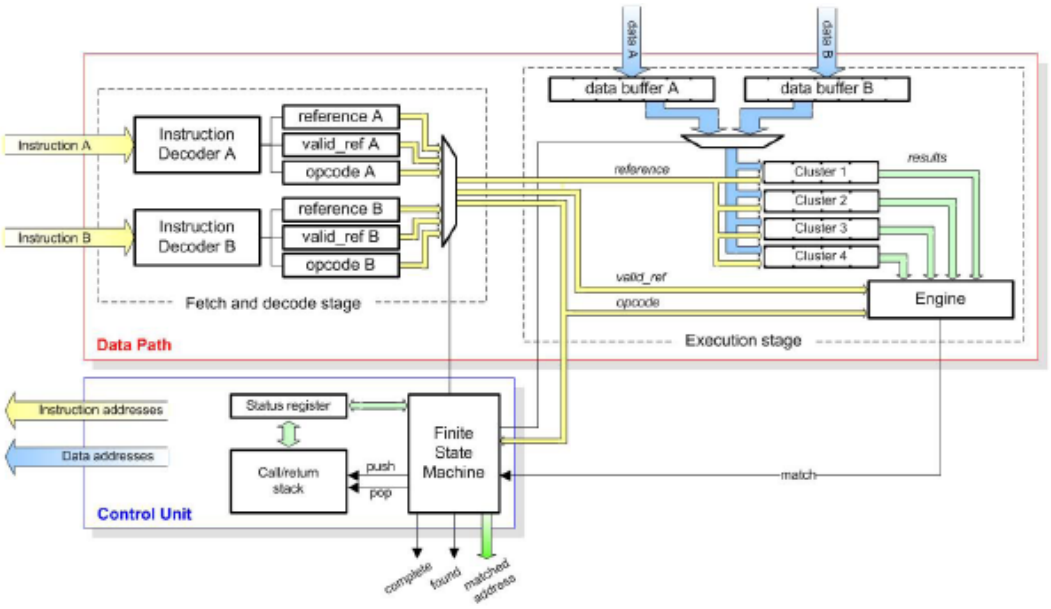
2.5.1 Překladač regulárního výrazu na kód

Překladač při překladu regulárního výrazu bere ohled na nastavitelnou VLIW architekturu procesoru a upravuje podle toho generovaný kód. Instrukce se skládá z krátkého operačního kódu a z vektoru referencí (počet referencí záleží na architektuře procesoru). Operátory +

a $*$ jsou implementovány jako smyčky přes instrukci, nebo skupinu instrukcí a závorky jsou implementovány jako volání funkce a tudíž je počet vnořených závorek omezen velikostí zásobníku návratových adres. Instrukce provádí porovnání textových dat s vektorem referencí.

2.5.2 Architektura

Procesor je založen na harvardské architektuře a skládá se ze dvou bloků - datové cesty a řídící jednotky. Datová cesta se skládá z dvoustupňové pipeline. Ve stupni fetch/decode se provádí načtení a dekódování instrukce. Tento stupeň obsahuje dvě jednotky pro načtení a dekódování instrukce. Pokud porovnávání selže, použije se stávající instrukce znovu, jinak se použije přednačtená instrukce z druhé jednotky. Ve stupni execute se provádí porovnávání v několika paralelních jednotkách. Délka jedné porovnávací jednotky určuje počet naráz zpracovávaných znaků. Každá následující paralelní jednotka je o 1 znak posunuta. S počtem paralelních jednotek se zvyšuje propustnost v případě, že regulární výraz není rozpoznán. Navržená architektura je na obrázku 2.15. Řídící jednotka řídí pipeline a řídí načítání a vykonávání instrukcí na základě stavu jednotky.



Obrázek 2.15: Architektura CPU pro efektivní vyhledávání RV (převzato z [4])

2.5.3 Výhody a nevýhody

Výhodou této metody je snadná modifikace vyhledávaných výrazů pouhým přehráním obsahu instrukční paměti. Nevýhodu je možné spatřovat v omezení maximálního počtu vnořených závorek velikostí zásobníku návratových adres a v tom, že čas potřebný pro zpracování jednoho znaku je nedeterministický, protože závisí na regulárním výrazu a zpracovávaném vstupním textu.

2.6 Shrnutí

Pro implementaci vyhledávání vzorů popsaných regulárními výrazy existuje celá řada algoritmů. Jejich vzájemné porovnání je však obtížné. Relativně snadné je porovnání přístupů založených na DFA a NFA z pohledu FPGA implementace. Porovnání v jednotlivých skupinách přístupů (NFA, DFA, ...) je však obtížné provést přesně. Studiem výše uvedených článků jsem došel k několika příčinám:

- Každý autor používá jinou technologii pro implementaci, čímž se do porovnání zanášejí vlivy závislé na dané technologii. Například běžně se používají 4 vstupé LUT, avšak technologie Virtex5 a Virtex6 používají 6 vstupé LUT, což vzájemné porovnání zkresluje.
- Každý autor používá jinou množinu regulárních výrazů, nebo jinou verzi dané množiny regulárních výrazů. Toto způsobuje to, že každý autor používá množinu jiné velikosti a struktury, což činí vzájemné porovnání obtížným.

Na příkladu tabulky porovnání v článku [17] je vidět, že se v ní vyskytuje pět různých architektur (4xFPGA, 1xASIC) a deset různých množin pravidel, což činí toto srovnání orientačním.

Proto jsem se rozhodl, že namísto programu pro generování jednotky jediného algoritmu navrhnu znovupoužitelnou a snadno rozšiřitelnou knihovnu, která bude umožňovat jednoduchou implementaci algoritmů pro vyhledávání vzorů, za účelem jejich snadného srovnání.

Pro účely implementace jsem zvolil hybridní přístup hledání deterministických částí 2.4.2 a pro porovnání jsem zvolil NFA přístup se sdíleným dekodérem znaků 2.2.1 a se sdílením znakových tříd 2.2.1.

Kapitola 3

Návrhy hardwarové struktury implementovaných jednotek

3.1 Platforma COMBOv2

Vývojovou platformu COMBOv2 používanou v projektu Liberouter tvoří několik typů HW karet s FPGA firmy Virtex a abstraktní vrstva NetCope umožňují rychlý vývoj aplikací pro tyto karty. Karty platformy COMBOv2 je možné rozdělit do dvou skupin na karty mateřské a karty rozhraní.

Mezi karty mateřské patří karta COMBO-LXT. Tato karta obsahuje FPGA Virtex5 LX110T/LX155T umožňující komunikaci po sběrnici PCI Express x8. Paměťový subsystém je tvořen dvěmi paměťmi typu QDRII SRAM o celkové kapacitě 144Mbit dosahující při frekvenci 250MHz maximální teoretické propustnosti 18Gb/s při zachování nízké latence. Dále je možné připojit až 2GB paměti DDR2 DRAM do slotu SO-DIMM umožňující dosáhnout při frekvenci 250MHz maximální teoretické propustnosti 32Gb/s. Komunikaci s kartou rozhraní umožňují dva vysokorychlostní konektory IFC, z nichž každý umožňuje dosáhnout každým směrem propustnosti až 28Gb/s. Karta dále obsahuje 4 konektory LSC, z nichž každý umožňuje dosáhnout každým směrem propustnosti až 4Gb/s. Design karty umožňuje změnu designu nahraného v FPGA za běhu bez nutnosti restartu počítače.

Mezi karty rozhraní patří například karty COMBOI-10G2 nebo COMBOI-1G4. Karta COMBOI-10G2 umožňuje připojit 2 spoje o rychlosti 10Gb/s. Karta COMBOI-1G4 umožňuje připojit 4 spoje o rychlosti 1Gb/s.

Vrstva NetCope pro rychlý vývoj aplikací na platformě COMBOv2 odstiňuje vývojáře od síťové komunikace a komunikace po sběrnici PCI-Express, poskytuje základní stavební bloky pro vývoj a poskytuje unifikované komunikační rozhraní. [10]

Čip Virtex 5 uvedený na trh firmou Xilinx poskytuje z hlediska vývoje jednotky pro vyhledávání vzorů v FPGA několik zajímavých vylepšení. Tento čip obsahuje nové šestivstupé vyhledávací tabulky (LUT). Každá LUT tak může implementovat logickou funkci o šesti vstupech, což snižuje počet spotřebovaných LUT a také může snížit délku logické cesty. Lze například implementovat 8-bitový komparátor hodnoty s pevným číslem v pouhých dvou LUT a tím pádem zkrátit délku logické cesty oproti implementaci LUT se 4 vstupy. Paměti BRAM umístěné na čipu mají nyní kapacitu 36Kbit přičemž zachovávají možnost dvouportového přístupu k nim. Paměť BRAM je možné také rozdělit na dvě samostatné paměti BRAM o kapacitě 18Kbit. Další informace o čipu Virtex 5 je možné najít v [18].

Tabulka 3.1: Vstupní rozhraní jednotky pro vyhledávání vzorů

Signál	Velikost	Popis
DATA	Šířka znaku	Vstupní data
SOF	1	Začátek datového rámce
EOF	1	Konec datového rámce
SRC_RDY	1	Vstupní data jsou platná
DST_RDY	1	Jednotka je připravena přijímat data

Tabulka 3.2: Výstupní rozhraní jednotky pro vyhledávání vzorů

Signál	Velikost	Popis
BITMAP	Počet RV	Nalezené výskyty RV jsou na příslušné pozici označeny 1
VLD	1	Bitmapa je platná
ACK	1	Potvrzení příjmu bimapy

3.2 Komunikační protokol rozhraní jednotek

Komunikační protokol rozhraní jednotek jsem navrhl tak, aby byl pro všechny jednotky společný. Tato vlastnost umožní snadnou zaměnitelnost a testování jednotek.

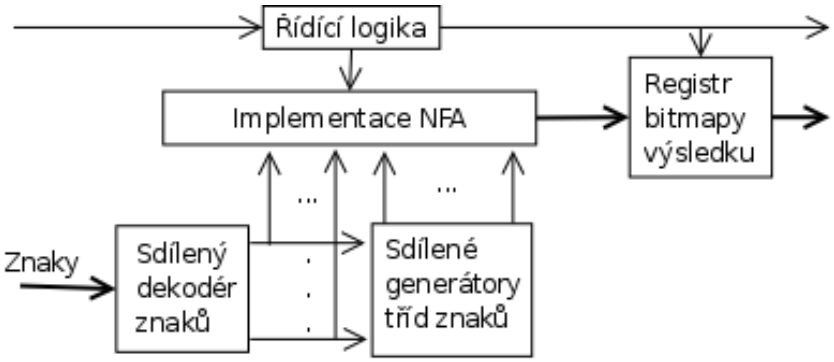
Vstupní rozhraní Vstupní rozhraní je popsáno v tabulce 3.1. Jednotka začíná zpracovávat data je-li signál SOF v logické jedničce a signál SRC_RDY je také v logické jedničce. Zpracování trvá dokud není za signálu SRC_RDY nastaveného do logické jedničky nastaven signál EOF také do logické jedničky včetně. Pokud je jednotka nepřipravena přijímat pak nastaví signál DST_RDY na logickou jedničku. Po dobu nastavení DST_RDY na logickou jedničku musí předcházející jednotka udržovat hodnoty SRC_RDY, SOF a EOF na stejné hodnotě do té doby než je jednotka připravena přijímat data. Tj. předcházející jednotka nastaví platnost dat bez ohledu na připravenost následující jednotky a jednotka nastaví připravenost přijímání dat bez ohledu na platnost dat předcházející jednotky. K přenosu dat tedy dochází pouze při současném nastavení signálů SRC_RDY a DST_RDY do logické jedničky. Jsou-li signály SRC_RDY a DST_RDY nastaveny do logické jedničky, pak je délka signálů SOF a EOF přesně 1 tak hodin. Výše uvedený protokol je po malých úpravách (inverze polarity signálů apod.) kompatibilní s protokoly FrameLink a základní variantou protokolu LocalLink.

Výstupní rozhraní Výstupní rozhraní je popsáno v tabulce 3.2. Jednotka průběžně nastavuje obsah registrů připojených na signál BITMAP. Po příchodu signálu EOF je nastaven signál platnosti bitmapy VLD na logickou jedničku. V tomto stavu zůstává dokud není přijato potvrzení příjmu zprávy následující jednotkou nastavením signálu ACK do logické jedničky. V následujícím taktu hodin je signál VLD nastaven na logickou nulu a jednotka může pokračovat v práci. V průběhu nastavení signálu VLD na logickou jedničku je nastaven signál DST_RDY vstupu jednotky na logickou nulu signalizující nepřipravenost jednotky přijímat další data.

3.3 Návrh hardwarové struktury jednotky používající NFA

Návrh vychází z přístupu uvedeného v kapitole o použití sdíleného dekodéru 2.2.1 s využitím sdílení tříd znaků uvedený v kapitole 2.2.1.

Blokové schéma jednotky je na obrázku 3.1. Jednotka se skládá z předřadného sdíleného dekodéru znaků, sdílených generátorů znakových tříd a vlastní logiky implementující části regulárních výrazů v HW. Pro implementaci předřadného sdíleného dekodéru znaků bude použita LUT implementace. Sdílené generátory znakových tříd budou implementovány jako generická hradla typu OR. Předřadný sdílený dekodér znaků se skládá z množství 8 bitových generických komparátorů.



Obrázek 3.1: Bloková struktura jednotky clark_nfa

Řídící logika je jednoduchá, má za úkol po dokončení přijmutí bloku dat, v němž se vyhledávání provádí, resetovat automat do výchozího stavu, nastavit příznak platnosti bitmapy výsledků a po příjetí potvrzení převzetí bitmapy povolit zpracování dalšího bloku dat a zresetovat bitmapu do výchozího stavu.

Bitmapa výsledků je implementována jako registr z RS klopných obvodů, do jehož každého bitu je možno nastavit samostatně logickou 1, resetován je celý naráz do logické 0. Obvod uchovává informaci o tom jaké regulární výrazy byly ve zpracovávaném bloku dat nalezeny.

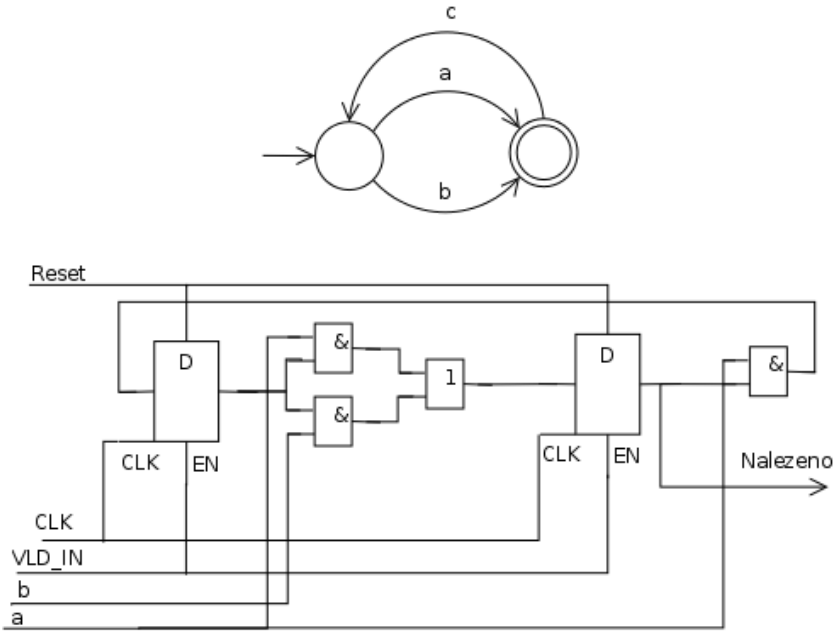
Samotná implementace NFA se skládá z D klopných obvodů, reprezentujících stav a z logiky reprezentující přechody. Pokud může více stavů provést přechod do jednoho následujícího stavu, pak jsou spojeny logickým operátorem OR. Samotný přechod se provede provedením logické operace AND mezi výstupem klopného obvodu reprezentujícího stav a signálem reprezentujícím symbol. Ukázka pro jednoduchý NFA je na obrázku 3.2.

Celá jednotka bude vzhledem ke své povaze generována softwarově.

3.3.1 Odhad zabraných zdrojů

Jelikož je délka znaku rovna osmi bitům, pak dekodér znaku zabere 2 LUT ve dvou úrovních logiky. Pokud je předřadný sdílený dekodér znaků implementován z LUT tak je pro něj potřeba 512 LUT, za předpokladu zpracovávání 1 znaku za takt. Pro jeden sdílený generátor znakových tříd (ekvivalentní operaci OR) počet úrovní logiky roven $u_l(N) = log_6(N)$, kde N je počet znaků ve třídě. Počet LUT zabraných generátorem pak zjistíme jako:

$$r[u_l(N) + 1] = N \tag{3.1}$$



Obrázek 3.2: Implementace KA s použitím Clarkova přístupu, nahoře odpovídající KA

$$x[i] = \lceil \frac{r[i+1]}{6} \rceil \text{ pro } i = u_l(N) \text{ až } 1 \tag{3.2}$$

$$r[i] = x[i] \tag{3.3}$$

$$g_l(N) = x[u_l] \tag{3.4}$$

Výsledná rovnice odhadu zabraných zdrojů logiky je pak:

$$L = 512 + \sum_{i=1}^{k} g_l(k_i) + S \tag{3.5}$$

kde k označuje počet tříd znaků, k_i počet znaků v dané třídě a S je velikost logiky zabrané samotnou implementací NFA, kterou můžeme spočítat jako:

$$S = \sum_{i=1}^{NFA_S} g_l(2 * I_i) \tag{3.6}$$

kde NFA_S je počet stavů NFA a I_i je počet přechodů končících ve stavu i.

Počet zabraných Flip-Flopů je možné odhadnout jako:

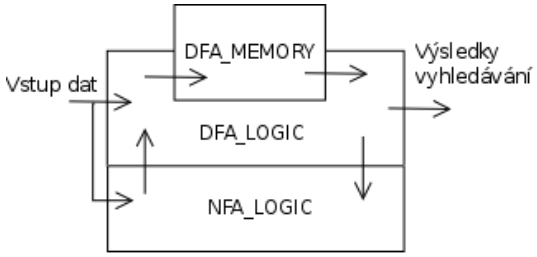
$$F = S + F \tag{3.7}$$

kde S je počet stavů v NFA části a F je celkový počet koncových stavů.

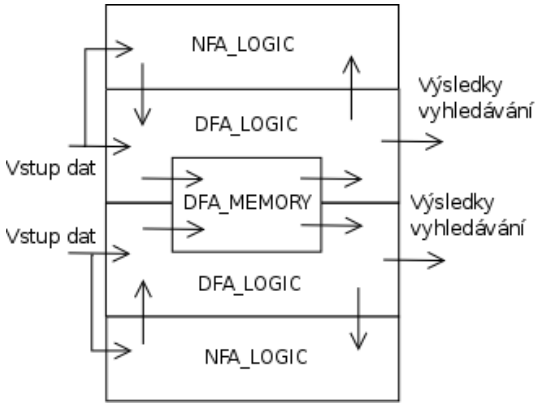
Výše uvedené odhady předpokládají FPGA čip Virtex 5, pro jiné technologie by je bylo třeba upravit.

3.4 Návrh hardwarové struktury jednotky Hledání deterministických částí

Návrh vychází z přístupu popsaného v 2.4.2. Jednotka je složena ze tří základních podkomponent. Jedná se o komponentu DFA_MEMORY implementující paměť tabulky přechodů DFA, dále se jedná o komponentu DFA_LOGIC implementující logiku DKA a o jednotku NFA_LOGIC implementující NFA. Celá jednotka se v závislosti na parametrech generování může skládat z různé kombinace těchto jednotek. Základní kombinace je zobrazena na obrázku 3.3. Jiná kombinace je znázorněna na obrázku 3.4, kdy dvě paralelní vyhledávací jednotky sdílí komponentu paměti.



Obrázek 3.3: Návrh hardwarové struktury jednotky



Obrázek 3.4: Návrh hardwarové struktury jednotky - dvě paralelní jednotky

3.4.1 Jednotka paměti tabulky přechodů

Tato jednotka DFA_MEMORY implementuje paměť tabulky přechodů deterministického konečného automatu. Je složena z pamětí typu BRAM, které jsou rychlé, mají malou latenci a jsou dvouportové. Dvouportovost pamětí BRAM dovoluje dvěma jednotkám pro vyhledávání vzorů sdílet jednu jednotku DFA_MEMORY. Jednotka je navržena genericky s ohledem na šířku požadovaného záznamu a na počet záznamů.

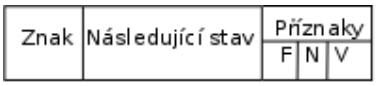
3.4.2 Jednotka logiky DFA

Tato jednotka DFA_LOGIC implementuje logiku deterministického konečného automatu. Schéma jednotky je na obrázku 3.7.

Struktura záznamu v tabulce přechodů

Struktura záznamu v tabulce přechodů je následující (viz obrázek 3.5):

- První pole obsahuje znak se kterým se provádí daný přechod - použije se na kontrolu přechodu v případě kolize.
- Druhé pole obsahuje číslo následujícího stavu.
- Třetí pole obsahuje příznaky přechodu. Jedná se o příznak koncového stavu, příznak toho, že se následující stav nachází v NFA a příznak toho, že je daný záznam tabulky platný.



Obrázek 3.5: Struktura záznamu tabulky přechodů

Implementace tabulky přechodů

Tabulka přechodů je implementována pomocí pamětí typu Block RAM (BRAM) v konfiguraci 2048 položek po 9 bitech. Minimální počet paralelních pamětí BRAM je tři. V této konfiguraci je k dispozici 8 bitů datových a jeden bit paritní v pro každou položku. Jelikož je možné paritní bit nastavit libovolně, je možné ho použít pro uložení dat. V tomto případě je použito datových bitů pro uložení čísla následujícího stavu a symbolu se kterým je tento přechod uskutečňován. Vlajky popsané výše jsou uloženy v paritních bitech.

Ukázka implementace tabulky přechodů pro délku čísla následujícího stavu 16-bitů, délky symbolu 8 bitů je na obrázku 3.6. Pro jednoduchost je předpokládán u této ukázky počet přechodů takový, aby se vlezl do tabulky o velikosti 2048 položek.

Optimalizace velikosti zápisových dekodérů do bitmapy

Pokud neexistuje z koncového stavu přechod, pak je možné do přechodové tabulky vložit místo čísla koncového stavu pořadové číslo regulárního výrazu ke kterému tento koncový stav patří a tím pádem ušetřit prostor na čipu, pokud je počet bitů pro reprezentaci celkového počtu regulárních výrazů výrazně menší než počet bitů potřebných pro reprezentaci celkového počtu stavů po namapování do implementované tabulky přechodů. Tento dekodér provádí dekódování z binárního kódu na kód 1 z n.

Bude-li například počet bitů reprezentace celkového počtu stavů roven 16 a počet bitů nutných pro reprezentaci celkového počtu regulárních výrazů bude 6 (předpokládejme 64 regulárních výrazů), pak bude ušetřeno 128 LUT, protože jeden dekodér pro 16 bitů zabere 3 LUT, zatímco jeden dekodér pro 6 bitů zabere 1 LUT.

Optimalizace velikosti dekodérů pro přechod do NFA části

Podobně jako pro u velikosti zápisových dekodérů do bitmapy můžeme podobnou optimalizaci zavést i pro velikost dekodérů pro přechod do NFA části.

Jelikož v případě přechodu do NFA části nemá číslo následujícího stavu v DFA části smysl, tak můžeme očíslovat cílové stavy v NFA části po pořadě a toto číslo uložit do pole určeného pro číslo následujícího stavu. Pokud bude počet bitů nutný pro reprezentaci počtu cílových stavů v NFA části výrazně nižší než počet bitů potřebných pro reprezentaci kódování z binárního kódu stavů na kód 1 z n, pak dojde k úspoře zdrojů.

Bude-li například počet bitů reprezentace celkového počtu stavů roven 16 a počet bitů nutných pro reprezentaci počtu cílových stavů v NFA části bude 11 (předpokládejme 2048 cílových stavů v NFA části), pak bude ušetřeno 2048 LUT, protože jeden dekodér pro 16 bitů zabere 3 LUT, zatímco jeden dekodér pro 11 bitů zabere 2 LUT.

Implementace vstupního dekodéru

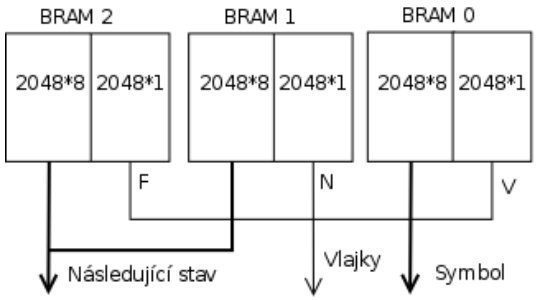
Vstupní dekodér provádí dekódování z kódu 1 z n na binární kód. Jsou možná dvě řešení, která budou v následujících odstavcích popsána. Výhodnost použití jednoho nebo druhého řešení záleží na tom, zda budeme chtít optimalizovat na velikost zabrané logiky na čipu FPGA nebo na velikost potřebné paměti pro zakódování přechodové tabulky. Implementován je druhý z navržených přístupů k implementaci vstupního dekodéru.

Standardní dekodér Tento přístup používá standardní dekodér z kódu 1 z n na binární kód, přičemž počet bitů pro reprezentaci binární hodnoty je roven $\lceil log_2(n) \rceil$. Tento přístup přináší malou velikost dekodéru, ale velikost paměti potřebné pro zakódování přechodové tabulky bude větší, protože pro bezkonfliktní zakódování přechodové tabulky bude třeba použít posun takto získaných čísel stavů doleva až o počet bitů nutný k zakódování celkového počtu možných znaků (např. 8 bitů pro ASCII), což je mnohem méně optimální než přístup popsaný v následujícím odstavci z důvodu ztížené práce heuristiky pro mapování přechodové tabulky.

Dekodér 1 z n na kódování stavů Tento přístup provádí dekódování z kódu 1 z n na kódování stavů, což je binární kód jehož počet bitů je větší než $\lceil log_2(n) \rceil$. Kódování stavů je určeno pomocí heuristiky pro mapování stavů přechodové tabulky, která se snaží najít takové očíslování stavů pro které dosáhne co nejmenší velikosti zabrané paměti. Tudíž tento přístup může mít větší velikost dekodéru než předchozí přístup. ale velikost potřebné paměti bude menší, protože jí nebude komplikovat práci pevné zakódování stavů jako u předchozího přístupu. Pro standardní dekodér z kódu 1 z n na binární kód platí, že má každý bit četnost výskytu jedničky rovnu 0.5, tudíž průměrná četnost jedničky je rovna 0.5. Pro dekodér na kódování stavů, platí že pravděpodobnost jedničky pro každý bit záleží na výsledku heuristiky a tudíž může být velikost dekodéru větší nebo menší než velikost standardního dekodéru v závislosti na četnosti výskytu jedničky v jednotlivých bitech výstupu.

Princip činnosti jednotky

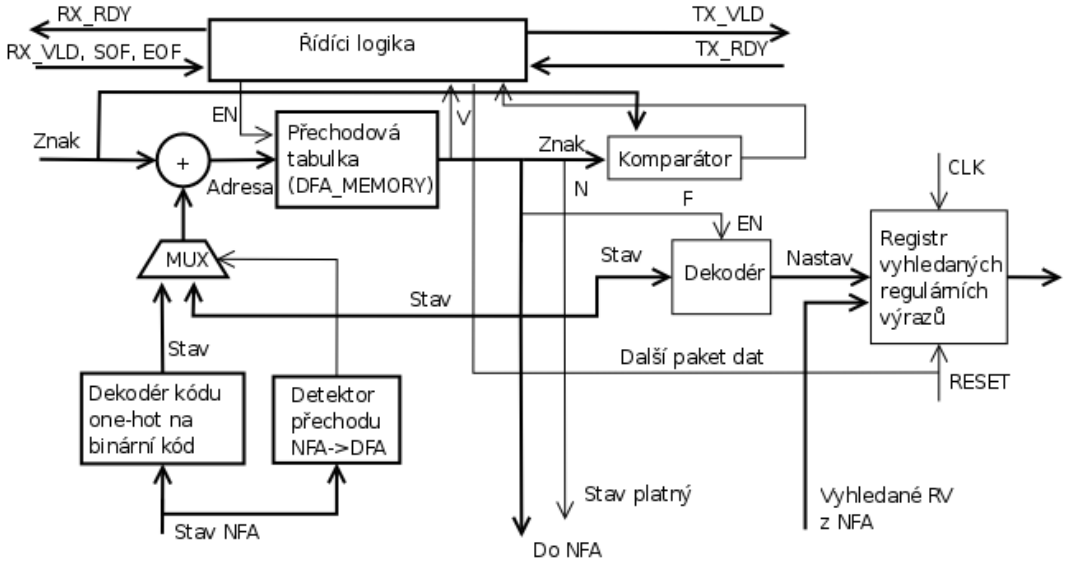
Pokud je jednotka v pasivním stavu pouze odebírá ze vstupu data a nekoná žádnou činnost. Pokud dojde k přechodu z NFA, pak převede reprezentaci stavů v jednotce NFA (one-hot) na reprezentaci v jednotce DFA, následně provede určení adresy v tabulce přechodů na základě současného stavu a vstupního symbolu. Načtený záznam je prozkoumán, je zjištěno zda je



Obrázek 3.6: Implementace tabulky přechodů

záznam platný (není-li platný je činnost jednotky zastavena do další aktivace z NFA), zda se jedná o koncový stav (pak je nastaven příslušný bit v registru výsledků) a zda se má přejít do NFA. Pokud se má přejít, přechod se provede a jednotka se zastaví do další aktivace z NFA, pokud se nemá provést přechod pokračuje činnost jednotky s tímto stavem.

Jednotka bude implementována genericky vzhledem k počtu řádků přechodové tabulky, datové šířce označení stavů, datové šířce řádku tabulky a počtu stavů vstupujících z NFA části.



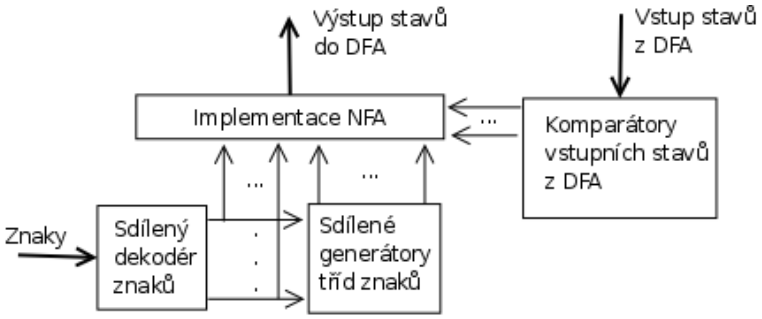
Obrázek 3.7: Návrh struktury jednotky DFA_LOGIC

3.4.3 Jednotka logiky NFA

Tato jednotka NFA_LOGIC implementuje logiku nedeterministického konečného automatu. Schéma jednotky je na obrázku 3.8. Jednotka se skládá z předřadného sdíleného dekodéru znaků, sdílených generátorů znakových tříd a vlastní logiky implementující části regulárních výrazů v HW. Pro přechod do vstupních stavů NFA jsou vygenerovány dekodéry z číselné reprezentace do selektivního signálu pro každý takový stav. Práce celé jednotky začíná v jednotce NFA, z ní se případně přechází do DFA.

Pro implementaci předřadného sdíleného dekodéru znaků bude použita BRAM, nebo LUT implementace. Sdílené generátory znakových tříd budou implementovány jako generická hradla typu OR.

Celá jednotka bude vzhledem ke své povaze generována softwarově.



Obrázek 3.8: Návrh struktury jednotky NFA_LOGIC

3.4.4 Odhad zabraných zdrojů

Pokud je rozsah čísel označujících stav menší nebo roven 2^{11}, pak takovýto dekodér zabere 2 LUT ve dvou úrovních logiky. Pokud je předřadný sdílený dekodér znaků implementován z LUT tak je pro něj potřeba 512 LUT, za předpokladu zpracovávání 1 znaku za takt. Pro jeden sdílený generátor znakových tříd (ekvivalentní operaci OR) počet úrovní logiky roven $u_l(N) = log_6(N)$, kde N je počet znaků ve třídě. Počet LUT zabraných generátorem pak zjistíme jako:

$$r[u_l(N) + 1] = N \tag{3.8}$$

$$x[i] = \lceil \frac{r[i+1]}{6} \rceil \text{ pro } i = u_l(N) \text{ až } 1 \tag{3.9}$$

$$r[i] = x[i] \tag{3.10}$$

$$g_l(N) = x[u_l] \tag{3.11}$$

Velikost logiky DFA je závislá na rozsahu čísel označující stav a příliš se nemění a označíme ji tedy jako empiricky zjistitelný parametr $C(n)$, kde n je rozsah čísel. Výsledná rovnice odhadu zabraných zdrojů logiky je pak:

$$L = 512 + 2 * s + \sum_{i=1}^{k} g_l(k_i) + S + C(n) \tag{3.12}$$

kde k označuje počet tříd znaků a k_i počet znaků v dané třídě, s je počet stavů vstupujících do NFA části z DFA části a S je velikost logiky zabrané samotnou implementací NFA (závislá na NFA), kterou můžeme spočítat jako:

$$S = \sum_{i=1}^{NFA_S} g_l(2 * I_i) \tag{3.13}$$

kde NFA_S je počet stavů NFA a I_i je počet přechodů končících ve stavu i.

Počet zabraných Flip-Flopů je možné odhadnout jako:

$$F = S + F + n \tag{3.14}$$

kde S je počet stavů v NFA části, n je počet zaráz zpracovávaných bitů (8,16,...), a F je celkový počet koncových stavů.

Počet zabraných BRAM je možné odhadnout jako:

$$B = \lceil \frac{T}{1024} \rceil * \lceil \frac{W}{36} \rceil \tag{3.15}$$

kde T je počet řádků přechodové tabulky, W je šířka záznamu v bitech.

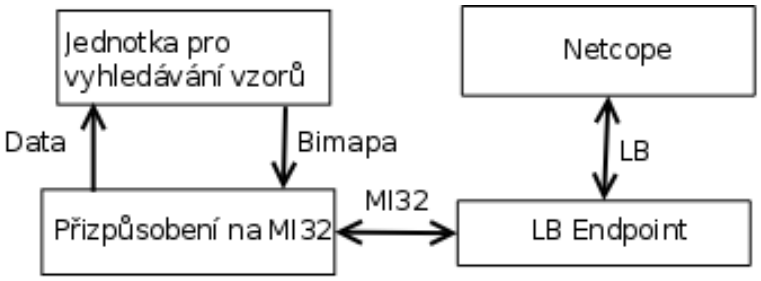
Výše uvedené odhady předpokládají FPGA čip Virtex 5, pro jiné technologie by je bylo třeba upravit.

3.5 Demonstrační implementace na platformě ComboV2

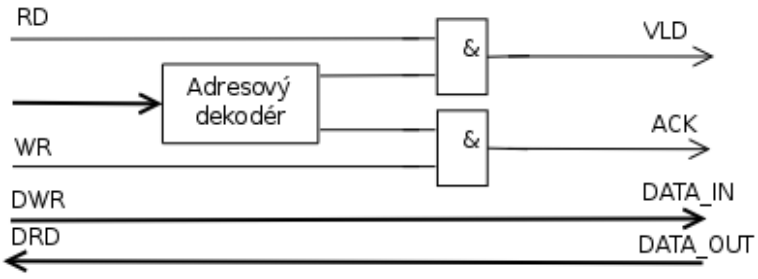
Za účelem ověření funkce na platformě FPGA jsem navrhl jednoduchý uživatelský design pro platformu urychlení návrhu NetCope, což je součást platformy ComboV2.

Implementace modulu se skládá z jednoduchého adresového dekodéru umožňujícího základní ovládání jednotky pro vyhledávání vzorů. Vstupní data jsou do komponenty postupně nahrávána přes lokální sběrnici pomocí utility csbus. Výsledek je možné si rovněž přečíst touto utilitou.

Na obrázku 3.9 je celková architektura modulu a na obrázku 3.10 je architektura adresového dekodéru.



Obrázek 3.9: Návrh struktury jednotky pro demonstraci funkčnosti na platformě ComboV2



Obrázek 3.10: Návrh struktury adresového dekodéru

Kapitola 4

Objektový návrh a implementace

Navrhovaný systém pro vyhledávání vzorů je součástí knihovny experimentů vyvíjené v rámci projektu ANT na Ústavu počítačových systémů Fakulty informačních technologií Vysokého učení technického v Brně.

Jednou z motivací pro vytvoření knihovny experimentů pro vyhledávání vzorů byla zjištění z kapitoly 2.6. Proto jsem při návrhu postupoval tak, aby byla knihovna použitelná pro různé přístupy k vyhledávání vzorů, byla snadno rozšiřitelná. Důraz byl kladen na modularitu řešení, snadnou testovatelnost knihovny experimentů po případných úpravách, vylepšeních a rozšířeních. Při návrhu a implementaci bylo použito prostředků objektově orientovaného programování.

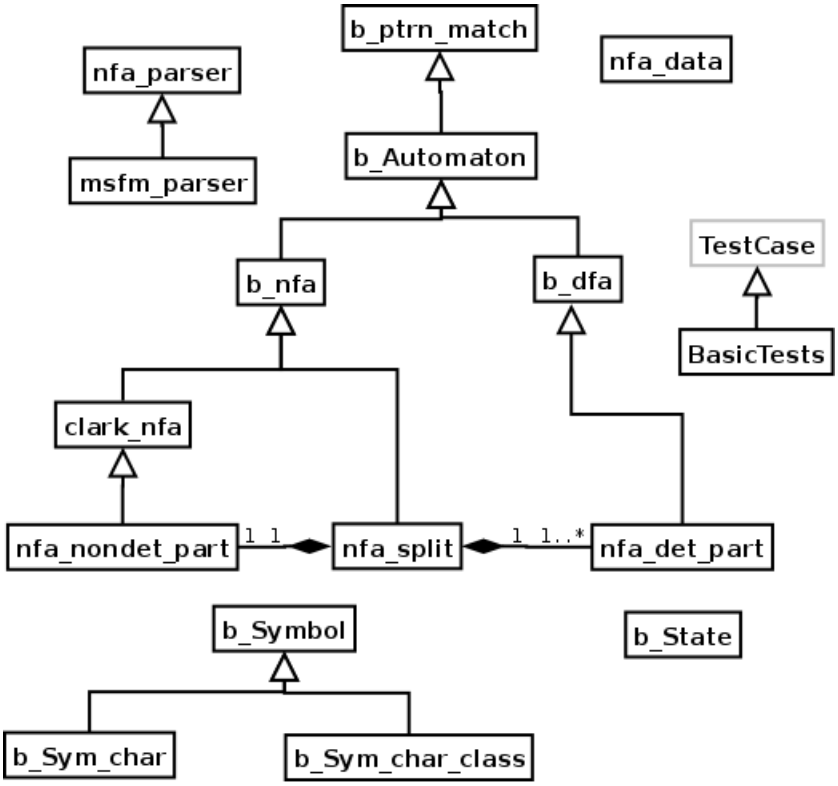
Pro implementaci byl vybrán skriptovací jazyk Python, konkrétně ve verzi 2.6. Tento skriptovací jazyk byl vybrán z důvodu snadného použití, vysoké rozšířenosti, množství poskytované funkcionality a rozšiřujících modulů a pokročilých vlastností pro vytváření testů. Nevýhodou skriptovacích interpretovaných jazyků obecně je jejich nízká rychlost v porovnání s jazyky kompilovanými. Tuto nevýhodu umožňuje Python po identifikování časově kritických částí snadno obejít, protože umožnuje používat moduly napsané v jazycich C/C++, což poskytuje potřebné zrychlení.

Na diagramu tříd 4.1 je navržená struktura třídní hierarchie implementace knihovny experimentů pro vyhledávání vzorů. Při návrhu je kladen důraz na důsledné používání dědičnosti, což přispívá k dobré modularitě, rozšířitelnosti a znovupoužitelnosti jednotlivých částí knihovny. V následujících podkapitolách jsou představeny jednotlivé třídy a jejich význam.

Jelikož v Pythonu není možné dědit soukromé metody a atributy tříd (tj. neexistuje analogie k protected v C++), byl zaveden úzus, že metody vnímané jako soukromé a děditelné se implementují jako metody veřejné s tím, že jejich název začíná pro odlišení jedním podtržítkem.

Při návrhu jsem rozdělil třídy do skupin podle jejich účelu. Základní rozdělení je na třídy pomocné implementující jednotlivé elementy konečných automatů (stavy, symboly), generování konečných automatů z regulárních výrazů, atd. a třídy algoritmů vyhledávání. U tříd algoritmů vyhledávání jsem provedl rozdělení na třídy implementující algoritmy vyhledávání vzorů a na bázové abstraktní třídy provádějící zobecnění společných vlastností. Takové rozdělení přispívá k výše uvedeným požadavkům na systém.

Pro vlastní generování HDL reprezentace jednotek pro vyhledávání vzorů pak slouží jednoduché skripty, které demonstrují jednoduchost a snadnost použití knihovny.



Obrázek 4.1: Celkový diagram tříd

4.1 Třída základní strukury automatu

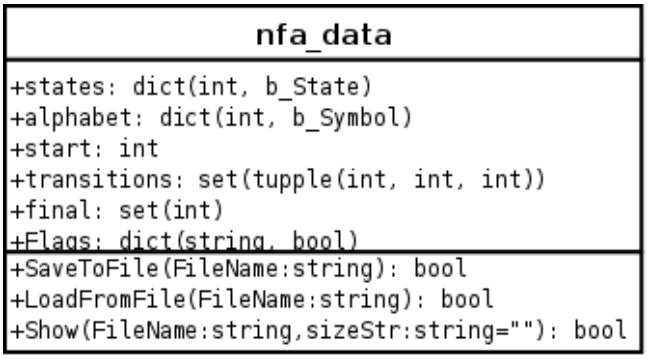
Tato třída nfa_data zobrazená na svém třídním diagramu 4.2 slouží jako základní třída zajišťující kompatibilitu a přenos dat konečného automatu mezi různým přístupy. Každý přístup dědící od třídy b_Automaton, popsané dále, musí být schopen načíst konečný automat z objektu třídy nfa_data a uložit konečný automat do objektu třídy nfa_data. Tato třída také poskytuje prostředku pro udržení datové persistence a umožnuje konečný automat zobrazit ve formátu GRAPHVIZ. Třída implementuje také standardní metody objektů Python __str__() a __repr__() umožnující vypsat obsah objektu třídy při použití funkce jazyka Python str().

Atributy Atributy třídy implementují formální model popsaný v kapitole 2.2.

states Slovník definující datovou strukturu konečné množiny stavů. Klíčem mohou být číselné hodnoty. Elementy jsou objekty tříd odvozené od bázové třídy b_State.

alphabet Slovník definující datovou strukturu konečné množiny symbolů vstupní abecedy. Klíčem mohou být číselné hodnoty. Elementy jsou objekty tříd odvozené od abstraktní bázové třídy b_Symbol.

start Číslo určující počáteční stav konečného automatu. Hodnota -1 označuje že automat nemá počáteční stav.



Obrázek 4.2: Diagram třídy nfa_data

Tabulka 4.1: Základní vlajky

Vlajka	Popis
Epsilon Free	Značí, že automat neobsahuje žádné epsilon přechody
Deterministic	Konečný automat je deterministický
CharClass Free	Konečný automat neobsahuje znakové třídy

transitions Množina obsahující přechody. Přechod je definován pomocí n-tice obsahující 3 prvky. První prvek obsahuje klíč do slovníku stavů reprezentující zdrojový stav. Druhý prvek obsahuje klíč do slovníku abecedy reprezentující vstupní symbol. Třetí prvek obsahuje klíč do slovníku stavů reprezentující cílový stav.

final Množina klíčů do slovníku stavů označující koncové stavy.

Flags Slovník reprezentující vlajky, které určují vlastnosti automatu. Klíči mohou být řetězce. Hodnotami může být hodnota libovolného typu, avšak standardní vlajky jsou logického datového typu.

Metody

SaveToFile(FileName) Uloží objekt do souboru. Pro implementaci je použita serializační knihovna jazyku Python cPickle.

LoadFromFile(FileName) Načte objekt ze souboru. Pro implementaci je použita serializační knihovna jazyku Python cPickle.

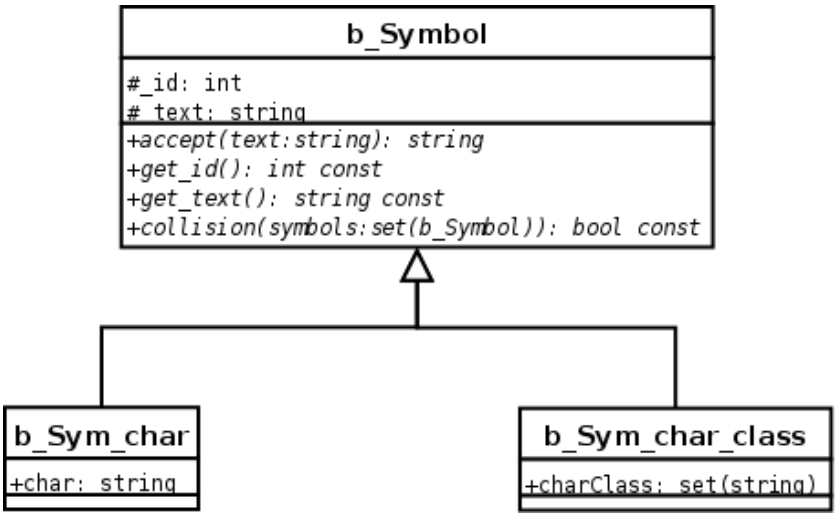
Show(FileName, sizeStr) Do souboru FileName uloží grafické zobrazení automatu ve formátu GRAPHVIZ. Volitelný parametr sizeStr umožňuje nastavit velikost výsledného grafu.

V tabulce 4.1 jsou shrnuty všechny základní vlajky používané ve vytvořených třídách.

4.2 Třídy reprezentace symbolů

Tato skupina tříd zahrnuje bázovou třídu b_Symbol a z ní odvozené třídy. Třídy jsou použity pro reprezentaci symbolů přechodů v konečném automatu. Všechny třídy musí být odvozeny

z bázové třídy. Diagram tříd pro tuto skupinu tříd je na obrazku 4.3. Všechny odvozené třídy také implementují standartní metody objektů Python __str__() a __repr__() umožnující vypsat obsah objektů tříd při použití funkce jazyka Python str(), metody __hash__(), __eq__() a __neq__() umožnující použít objekty těchto tříd jako klíče slovníků a prvky množin, což zvyšuje jejich použitelnost. Nevýhodu tohoto přístupu lze spatřovat v tom, že s objekty se pak musí zacházet jako s neměnnými, což v praxi nečiní problémy.



Obrázek 4.3: Diagram tříd symbolů

4.2.1 Bázová třída symbolu

Bázová abstraktní třída b_Symbol, definuje jednotné komunikační rozhraní pro všechny z ní odvozené třídy symbolů. V rámci této bázové třídy jsou definovány metody pro získání identifikace a textové reprezentace symbolu, zjištění kolize s jinými symboly a pro příjem symbolu ze řetězce přijmutého jako parametr. Funkce pro příjem vrací text bez prijatého znaku, není-li možné počáteční znak symbolem přijmout je vyvolána vyjímka.

Atributy

_id Identifikačné číslo symbolu. Musí korespondovat s odpovídající hodnotou klíče slovníku vstupní abecedy.

_text Textová reprezentace symbolu. Použitá je při vykreslování grafické reprezentace konečného automatu.

Metody

get_text() Vrací textovou reprezentaci symolu pro vykreslování grafické reprezentace konečného automatu.

get_id() Vrací identifikační číslo symbolu.

accept(text) Je-li možné přijmou první znak řetězcové proměnné text za použití symbolu reprezentovaného objektem této třídy, pak vrátí řetězec bez tohoto znaku, v opačném případě vyvolá vyjímku.

collision(set_of_symbols) Vrací logickou hodnotu pravda, došlo-li ke kolizi mezi symbolem reprezentovaným objektem této třídy a některým z množiny objektů z proměné set_of_symbols. V opačném případě vrací pravdivostní hodnotu nepravda.

4.2.2 Třída znaku

Tato třída b_Sym_char odvozená od bázové abstraktní třídy b_Symbol implementuje symbol jako jeden znak ASCII. Tato třída také implementuje epsilon, jež je reprezentován jako prázdný řetězec.

Atributy

char Znakový symbol ve formátu ASCII. Reprezentován je řetězcem obsahujícím nejvýše jeden znak. Řetězcem nulové délky je implementován symbol epsilon.

4.2.3 Třída znakové třídy

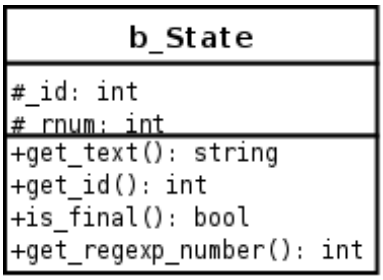
Tato třída b_Sym_char_class odvozená od bázové abstraktní třídy b_Symbol implementuje symbol jako třídu znaků. Třída znaků je implementována jako množina znaků.

Atributy

charClass Množina znaků ve formátu ASCII. Každý prvek množiny je reprezentován jako řetězec délky právě jedna.

4.3 Třída reprezentace stavů

Tato bázová třída b_State implementuje reprezentaci stavu konečného automatu. V případě potřeby rozšířené funkcionality je možné provést odvození od této třídy. Jednotka obsahuje metody pro zjištění čísla stavu, čísla regulárního výrazu a či je stav koncový. Třída implementuje také standardní metody objektů Python __str__() a __repr__() umožnující vypsat obsah objektu třídy při použití funkce jazyka Python str(). Diagram této třídy je na obrázku 4.4.



Obrázek 4.4: Diagram třídy b_State

Atributy

_id Identifikační číslo stavu. Toto číslo se musí shodovat s klíčem slovníku stavů odkazujícího na objekt této třídy.

_rnum Je-li stav koncový, pak je v tomto stavu číslo odpovídajícího regulárního výrazu. Není-li stav koncový, pak je v něm uloženo číslo -1.

Metody

get_text() Vrací textovou reprezentaci symbolu pro účely vykreslení grafické reprezentace grafu.

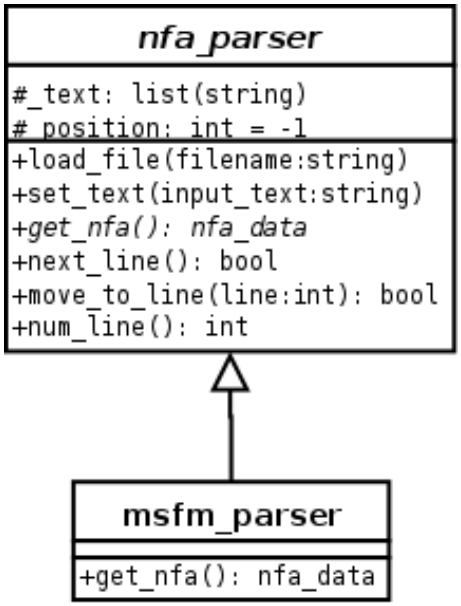
get_id() Vrací identifikační číslo stavu.

is_final() Vrací logickou hodnotu pravda, je-li stav koncový. V opačném případě vrací logickou hodnotu nepravda.

get_regexp_number() Vrací pořadové číslo regulárního výrazu odpovídající tomuto koncovému stavu.

4.4 Třídy zpracování regulárních výrazů

Diagram tříd zpracování regulárních výrazů je na obrázku 4.5. Tyto třídy slouží pro převod regulárních výrazů na nedeterministické konečné automaty.



Obrázek 4.5: Diagram tříd zpracování regulárních výrazů

4.4.1 Bázová třída zpracování regulárních výrazů

Tato abstraktní bázová třída zpracování regulárních výrazů nfa_parser definuje komunikační rozhraní pro všechny odvozené třídy zpracování regulárních výrazů.

Práce s těmito třídami probíhá tak, že se nejdříve nahrají regulární výrazy ze souboru nebo se načtou z řetězce. Následně je možné se pomocí funkcí pro pohyb po načtených regulárních výrazech dostat na konkrétní výraz a ten převést na konečný automat reprezentovaný objektem třídy nfa_data.

Atributy

_text Seznam regulárních výrazů. Každý regulární výraz je reprezentován řetězcem.

_position Ukazatel na aktuálně zpracovávaný výraz.

Metody

load_file(filename) Načte soubor obsahující regulární výrazy. Jméno souboru je v parametru metody filename.

set_text(input_text) Načte řetězec předaný parametrem input_text jako množinu regulárních výrazů.

get_nfa() Provede převod aktuálního regulárního výrazu na nedeterministický konečný automat. Vrací objekt třídy nfa_data.

next_line() Provede přechod na následující regulární výraz a vrátí v případě úspěchu logickou hodnotu pravda a v případě pokusu o přechod za poslední regulární výraz vrací logickou hodnotu nepravda.

move_to_line(line) Provede přechod na regulární výraz s číslem odpovídajícím parametru line a vrátí v případě úspěchu logickou hodnotu pravda a v případě pokusu o přechod mimo minimální nebo maximální číslo regulárního výrazu vrací logickou hodnotu nepravda.

num_lines() Vrací počet načtených regulárních výrazů.

4.4.2 Třída zpracování automatu ve formátu MSFM

Tato třída msfm_parser tvoří obálku nad nástrojem pro převedení regulárních výrazů na konečný automat, který byl vyvinut na UPSY. Tento nástroj vytváří soubory ve formátu MSFM, které tato třída zpracovává a vytváří z nich objekt třídy nfa_data. Tento nástroj nepodporuje vícenásobné paralelní spuštění.

Formát MSFM

Na prvním řádku je počet stavů, na druhém řádku je uveden počet přechodů konečného automatu. Na následujících řádcích jsou uloženy jednotlivé přechody. Na každém řádku je jeden záznam přechodu formátu: zdrojový stav, číslo symbolu, cílový stav a příznak epsilon přechodu, oddělených svislítkem. Konec záznamů přechodů je označen řádkem mřížek. Následuje řádka na níž jsou uvedeny všechny koncové stavy. Každý stav je oddělen čárkou. Konec záznamů koncových stavů je označen řádkem mřížek. Následují až do konce souboru

záznamy jednotlivých symbolů. Každý symbol je uložen na samostatném řádku a má formát číslo symbolu: znak||znak||...||. Jedná se textový formát.

4.5 Třídy přístupů

Tato skupina tříd obsahuje jak bázové abstraktní třídy pro implementované přístupy k vyhledávání vzorů, tak odvozené třídy implementující jednotlivé algoritmy pro vyhledávání vzorů.

4.5.1 Bázová třída vyhledávání vzorů

Tato abstraktní bázová třída b_ptrn_match obsahuje metody společné všem přístupům k vyhledávání vzorů. Jedná se metody zjištující spotřebovanou paměť, množství zabrané logiky čipu FPGA a metodu pro nalezení vzorů v zadaném řetězci.

Atributy

_consumed_memory Udává velikost spotřebované paměti pro daný algoritmus rychlého vyhledávání vzorů. Velikost paměti je udána v bytech.

_consumed_logic Udává velikost spotřebované logiky pro daný algoritmus rychlého vyhledávání vzorů. Implemntován je tento parametr jako n-tice o třech prvcích. První prvek udává počet použitých LUT, druhý prvek udává počet použitých klopných obvodů a třetí prvek udává počet použitý BRAM. Odvozené třídy mohou způsob uložení parametrů změnit.

Metody

report_memory() Vrací množství spotřebované paměti v bytech.

report_logic() Vrací množství spotřebované logiky.

search(input_string) Funkce provede vyhledání vzorů v řetězci daném parametrem input_string za použití daného přístupu.

4.5.2 Bázová třída algoritmů založených na konečném automatu

Bázová abstraktní třída b_Automaton odvozená od abstraktní bázové třídy b_ptrn_match slouží jako základ pro přístupy k vyhledávání vzorů popsaných regulárními výrazy pomocí konečných automatů. Implementuje také zděděnou metodu search() pro konečné automaty.

Atributy

_automaton Datová struktura reprezentující konečný automat. Jedná se o objekt třídy nfa_data.

Metody

create_from_nfa_data(nfa) Přiřadí do atributu _automaton hlubokou kopii parametru nfa. Parametr nfa musí být objekt třídy nfa_data.

create_by_parser(nfa_parser_class) Tato metoda vytvoří objekt třídy nfa_data na základě objektu třídy nfa_parser předaného v parametru nfa_parser_class. Vytvořený objekt je pak uložen do atributu _automaton. Obsahuje-li objekt třídy nfa_parser více než jeden regulární výraz, pak je výsledný vytvořený objekt sjednocením konečných automatů pro všechny regulární výrazy.

get_automaton() Vrátí hlubokou kopii parametru _automaton.

join(nfa, modify_reg_exp_num) Tato metoda provede sjednocení konečného automatu reprezentovaného atributem _automaton s objektem třídy nfa_data předaným v parametru nfa. Nepovinný parametr modify_reg_exp_num určuje zda se má provést modifikace čísel regulárních výrazů v koncových stavech tak, aby bylo výsledné číslování konzistentní. Sjednocení je provedeno tak, aby nevznikala redundance abecedy a aby bylo číslování stavů konzistentní. Vlastní sjednocení je pak po těchto úpravách provedeno přidáním epsilon přechodu z počátečního stavu prvního automatu do počátečního stavu druhého automatu. Výchozí hodnotou je logická hodnota nepravda.

Nuluje vlajky Deterministic a Epsilon Free.

_epsilon_closure(state, StateOutSymbols) Pro stav zadaný parametrem state vypočítá epsilon uzávěr. Parametr StateOutSymbols obsahuje pomocnou strukturu pro urychlení výpočtu.

remove_epsilons() Provede pro daný konečný automat odstranění epsilon přechodů. Využívá výše uvedenou metodu pro výpočet epsilon uzávěru. Nastavuje vlajku Epsilon Free.

remove_unreachable() Tato metoda provede odstranění stavů, které jsou izolované, nedosažitelné nebo z nich nevede žádná sekvence přechodů do nějakého koncového stavu.

_removeCharClasses() Metoda provede odstranění znakových tříd vytvořených při převodu regulárních výrazů na konečný automat. Přechod, jehož vstupním symbolem je třída znaků, je nahrazen množinou přechodů jejichž vstupními symboly jsou znaky původní třídy znaků.

_createCharClasses() Metoda provede vytvoření tříd znaků. Třída znaků je vytvořena tehdy, existuje-li více přechodů ze stejného zdrojového stavu do cílového stavu. Tyto přechody jsou pak odstraněny a je vytvořen přechod jehož vstupním symbolem je třída znaků složená ze vstupních symbolů odstraněných přechodů.

4.5.3 Bázová třída algoritmů založených na NFA

Tato bázová abstraktní třída b_nfa definuje rozhraní specifické pro přístupy založené na nedeterministickém konečném automatu. Účelem této metody je sdružovat optimalizace a metody specifické pro nedeterministické konečné automaty.

Tabulka 4.2: Parametry genetického algoritmu

Parametr	Výchozí hodnota	Popis
populationSize	10	Velikost populace.
generations	1000	Maximální počet generací.
mutationProbablity	0.3	Pravděpodobnost, že se použije pouze mutace.
mutationRate	0.5	Pravděpodobnost mutace kříženého jedince.
mutationCount	5	Počet prováděných mutací genomu v jedné generaci.

4.5.4 Bázová třída algoritmů založených na DFA

Tato bázová abstraktní třída b_dfa definuje rozhraní specifické pro přístupy založené na deterministickém konečném automatu. Účelem této metody je sdružovat optimalizace a metody specifické pro deterministické konečné automaty.

Tato třída také implementuje metody pro namapování přechodové tabulky definované pro algoritmus hledání deterministických částí v kapitole 2.4.2. Tyto metody jsou zde implementovány proto, že mohou být použitelné pro mapování přechodové tabulky i u jiných algoritmů pro vyhledávání vzorů založených na DFA.

Atributy

_state_representation Tento atribut obsahuje seznam kompozitních jmen stavů deterministického konečného automatu. Pro každý stav DFA je zde uložena množina stavů nedeterministického konečného automatu, ze kterého daný stav při determinizaci vznikl.

Metody

determinise(create_table) Tato metoda provede determinizaci konečného automatu. Pro determinizaci jsou podporovány symboly typu znak a třída znaků. Vstupní automat nesmí obsahovat epsilon přechody. Nepovinný parametr create_table značí, že se nemá uchovávat obsah atributu _state_representation, což vede k úspoře paměti. Počáteční hodnota tohoto parametru je neuchovávat obsah. Pro svoji činnost využívá následující funkci _allIntersections(). Nastavuje vlajku Deterministic.

_allIntersections(sets) Tato metoda vypočítá všechny možné průsečíky seznamu symbolů. Každý prvek seznamu je reprezentován množinou, což umožňuje práci se třídami znaků. Seznam symbolů je předán v parametru sets. Vrací seznam všech průsečíků. Při výpočtu se postupuje tak, že pokud je znak z množiny použit v průsečíku, pak je z množiny odstraněn.

getTableByGA(maxSize, parametryGA) Tato metoda se snaží najít mapování tabulky přechodů na její hardwrovou implementaci pomocí genetického algoritmu popsaného v 2.4.2. Výsledkem je mapování původního označení stavů na vypočítané označení stavů. Toto mapování je uloženo pomocí datového typu slovník. Není-li dosaženo úspěchu, pak je vrácena hodnota None. Vysvětlení významu a počáteční hodnoty parametrů genetického algoritmu (zapsány zkráceně jako parametryGA) jsou v tabulce 4.2. Parametr maxSize udává maximální povolenou velikost paměti. Algoritmus se v případě nalezení řešení snaží optimalizovat velikost zabrané paměti.

_evaluate(parametry) Metoda provede vyhodnocení fitness funkce pro celou populaci.

_rank(rank, ssum) Metoda provádí výběr jedince pomocí tzv. ranku. Pro každého jedince je určen na základě hodnoty fitness funkce rank (předán v parametru rank). Následně je vygenerována náhodná hodnota v rozmezí nula až součet všech hodnot ranků. Vrácen je index prvku pro kterého přesáhl součet ranků hodnotu náhodného čísla předtím vygenerovaného.

_create(parametry) Na základě hodnoty fitness funkce a s pomocí selekce pomocí ranku vytvoří podle zadaných parametrů novou generaci. Je uplatněn elitismus a nejlepší jedinec tedy přežívá do další generace. Je-li nejlepších jedinců více je vybrán nejmladší z nich.

getTableByHeuristic() Tato metoda vypočítá mapování tabulky přechodů na její hardwarovou implementaci podle heuristiky popsané v 2.4.2. Výsledkem je dvojice, kdy první prvek obsahuje mapování původního označení stavů na vypočítané označení stavů. Toto mapování je uloženo pomocí datového typu slovník. Druhý prvek obsahuje minimální potřebnou velikost paměti v bytech potřebnou pro hardwarovou reprezentaci tabulky přechodů. Heuristika se snaží najít co nejoptimálnější mapování stavů na co nejmenší výslednou velikost paměti.

4.5.5 Třída algoritmu sdílení dekodérů a znakových tříd

Třída clark_nfa implementuje algoritmus převodu nedeterministického konečného automatu na odpovídající hardwarovou implementaci. Jsou využity následující optimalizace:

- Sdílený dekodér znaků z kódu binárního na kód 1 z n podle 2.2.1
- Sdílení tříd znaků podle 2.2.1. Sdílení tříd znaků je možné vypnout pomocí použití metody _removeCharClasses(), která odstraní všechny třídy znaků.

Atributy

width Datová šířka vstupního symbolu v bitech. Zatím podporována datová šířka 8 bitů odpovídající jednomu znaku ve formátu ASCII.

template Udává název souboru s VHDL šablonou jednotky.

_statistics Slovník statistických informací o vytvořené jednotce.

_useBram Indikuje zda se mají pro konstrukci adresového dekodéru použít paměti typu BRAM. Zatím není tento typ adresového dekodéru použit.

_LUTInputs Udává počet vstupů logického elementu FPGA LUT. Závisí na použité technologii FPGA. Pro většinu typů je obvyklá hodnota 4, pro typy Virtex 5 a Virtex 6 firmy Xilinx je typická hodnota 6.

Metody

get_HDL() Vygeneruje kód v jazyce VHDL implementující jednotku pro vyhledávání vzorů, která používá algoritmus pro rychlé vyhledávání vzorů se sdíleným dekodérem a se sdílením tříd znaků. Pro svoji činnost používá níže uvedené metody.

Tabulka 4.3: Statistiky poskytované v atribut _statistic pro třídu nfa_split.

Statistika	Význam
NFA States	Počet stavů nedeterministického konečného automatu.
NFA Transitions	Počet přechodů nedeterministického konečného automatu.
DFA States	Počet stavů deterministického konečného automatu.
DFA Transitions	Počet přechodů deterministického konečného automatu.
NCA Sizes	Seznam velikostí nalezených bezkolizních množin.
N Size	Velikost kolizní množiny.
DFA Parts States	Seznam počtu stavů deterministických částí.
DFA Parts Transitions	Seznam počtu přechodů deterministických částí.
NFA Part States	Počet stavů nedeterministické části.
NFA Part Transitions	Počet přechodů nedeterministické části.
NFA2DFAs Target states	Počet stavů z nichž existuje výstupní přechod do deterministických částí.

_get_char_dec_HDL() Vygeneruje VHDL kód pro sdílený dekodér znaků. Dekódování se provádí z kódu binárního na kód 1 z n.

_get_logic_HDL() Vygeneruje VHDL kód vnitřní logiky stavů a přechodů automatu. Pro výpočet se využívá pomocné struktury obsahující zpětné přechody uložené ve formě slovníku.

_get_final_HDL() Vygeneruje VHDL kód pro propojení koncových stavů s registrem výsledné bitmapy.

4.5.6 Třídy algoritmu hledání deterministických částí

Tato skupina tříd obsahuje třídy algoritmu hledání deterministických částí.

Třída nfa_split

Třída nfa_split zastřešuje celý algoritmus a vytváří objekty následujících tříd implementující jednotlivé části jednotky.

Atributy

_det_parts Seznam objektů třídy deterministické části jednotky. Počet objektů je roven počtu vygenerovaných deterministických částí NFA.

_nfa_part Objekt třídy nedeterministické části jednotky.

_statistics Slovník statistických informací o vytvořené jednotce. Přehled informací a jejich významu je v tabulce 4.3.

Metody

create_det_parts(num, method) Vytvoří pro daný nedeterministický konečný automat odpovídející deterministické části a nedeterministickou část pomocí algoritmu hledání deterministických částí. Je možné zvolit počet hledaných deterministických částí parametrem num. Výchozí hodnota tohoto parametru je 1. V případě, že je požadovaný

počet deterministických částí větší než je vytvořitelný počet částí, je vytvořen vytvořitelný počet částí. Parametr metod udává metody mapování tabulky přechodů na její implementaci. Výchozí hodnota tohoto parametru je 0, což značí použití heuristiky. Další povolená hodnota je 1 značící použití experimentálního genetického algoritmu. Pro každou část je také vypočítána množina vstupních a výstupních stavů z dané části. V případě neúspěchu vrací logickou hodnotu nepravda a v případě úspěchu vrací logickou hodnotu pravda.

_GetCollisionCard(ModstatesCollisions) Vrací kardinalitu elementů slovníku překládajícího stav DFA na odpovídající stavy NFA z nichž se stav skládá.

_GetRemovedCollisions(state, ModstatesCollisions) Heuristika číslo dvě určuje kolik kolizí bude odstraněno, pokud se odstraní stav zadaný parametrem state.

_minSet(collisionCard) Nalezne množinu stavů s minimálním počtem kolizí. Jedná se o heuristiku číslo jedna.

_findMaximumHeur(minimalSet, ModstatesCollisions) S použitím heuristiky číslo dvě najde v množině stavů s minimálním počtem kolizí nejlepší stav (tj. stav jehož odstraněním je maximálně snížen počet kolizí). Je-li nejlepších stavů více, vybere se jeden z nich.

_NonCollisionSet(ModstatesCollisions) Vypočítá množinu všech stavů, které nejsou vzájemně v kolizi. Pro svoji funkci používá výše uvedené metody.

_AllNonCollisionSets(ModstatesCollisions) Vypočítá všechny množiny kolizních stavů. Stavy v rámci množiny v kolizi nejsou, ale množiny vzájemně v kolizi jsou. Používá se při dělení na více deterministických částí.

Třída nedeterministické části

Tato třída nfa_non_det_part implementuje nedeterministickou část automatu. V podstatě pouze rozšiřuje třídu clark_nfa o vstupy a výstupy z deterministických částí.

Atributy

_inset Množina stavů z nichž existuje přechod do jednotky.

_outset Množina stavů z nichž existuje přechod z jednotky.

Metody

get_HDL() Vygeneruje kód v jazyce VHDL implementující nedeterministickou část jednotky pro vyhledávání vzorů, která používá algoritmus hledání deterministických částí.

_get_io_HDL() Vygeneruje VHDL kód řídící vstup a výstup z jednotky směrem od a k deterministickým částem.

Třída deterministické části

Třída nfa_det_part implementuje deterministickou část automatu. Rozšiřuje třídu b_dfa o generování výsledného HDL kódu vlastní jednotky, generování HDL kódu paměťové subkomponenty a generování HDL kódu vstupních a výstupních dekodérů. Jednotka používá pro generování HDL kódu komponent šablony, které obsahují části kódu komponenty nezávislé na vygenerovaných částech.

Atributy

_inset Množina stavů z nichž existuje přechod do jednotky.

_outset Množina stavů z nichž existuje přechod z jednotky.

_index Číslo deterministické jednotky. V případě vygenerování více deterministických částí jsou podle tohoto čísla rozlišeny jednotlivé jednotky.

Metody

starting_states() Vratí množinu vstupních stavů (tj. stavů z nichž existuje přechod do jednotky).

out_states() Vrátí množinu výstupních stavů (tj. stavů z nichž existuje přechod z jednotky).

ttable_size() Vypočítá velikost přechodové tabulky deterministické části.

ttable_min() Vypočítá minimální velikost tabulky v bytech. Předpokládá nejmenší možnou reprezentaci tabulky. Jedná se pouze o odhad. Reálná minimání dosažitelná velikost může být větší.

ttable_overlap() Vrátí velikost paměti v bytech potřebnou pro zakódování přechodové tabulky při použití heuristiky nebo GA.

ttable_overlap_BRAMS(bram_size, bram_word) Vrátí počet pamětí BRAM potřebných pro zakódování přechodové tabulky při použití heuristiky nebo GA. Parametry bram_size a bram_word je popořadě udána velikost paměti v řádcích a délka jednoho řádku paměti v bitech.

encoder_size() Vypočítá počet LUT potřebných pro implementaci vstupního dekodéru.

encoder_HDL() Vygeneruje HDL popis jednotky implementující vstupní dekodér.

decoder_size() Vypočítá počet LUT potřebných pro implementaci výstupního dekodéru.

decoder_HDL() Vygeneruje HDL popis jednotky implementující výstupní dekodér.

memory_HDL(wrap, bram_size, bram_word) Vygeneruje HDL popis jednotky reprezentující přechodovou tabulku automatu. Parametr wrap určuje zda se bude brát v úvahu cykličnost vytvořené paměti nebo ne, parametry bram_size a bram_word je popořadě udána velikost paměti v řádcích a délka jednoho řádku paměti v bitech. Jednotka se skládá z pamětí typu BRAM obsahující přechodovou tabulku automatu. Pro svoji činnost používá následující metodu.

_map2memory() Metoda provede namapování části přechodové tabulky o velikosti odpovídající jedné paměti BRAM do potřebného počtu pamětí BRAM v závislosti na šířce paměti BRAM. K uložení obsahu paměti jsou použity inicializační vektory jednotky paměti BRAM.

detunit_HDL() Vygeneruje HDL popis vlastní jednotky deterministické části automatu.

4.6 Třídy testování

Do této skupiny patří třída BasicTests která dědí od třídy TestCase poskytované modulem jazyka Python unittest. Tento modul poskytuje prostředky pro snadné a efektivní vytváření automatických testů. Vytvořená třída BasicTests poskytuje 32 různých testů vytvořených tříd.

4.7 Možnosti rozšíření

Možnosti budoucího rozšíření jsou v několika oblastech:

- Podpora zpracování více znaků v jednom hodinovém cyklu, což přináší zvýšení dosažitelné propustnosti implementací.
- Další oblastí je podpora jiných způsobů zápisu regulárních výrazů než je používaný PCRE. Jedná se například o podporu zpracování kompletních pravidel IDS systému Snort a Bro.
- Dalším zamýšleným rozšířením je implementace dalších algoritmů rychlého vyhledávání vzorů.
- Zvýšení výkonnosti algoritmů přepsáním kritických částí do jazyka C.
- Použití tříd znaků pro algoritmy založené na DFA a pro algoritmy založené na NFA použít tuto optimalizaci v předřadném dekodéru.

Kapitola 5

Výsledky

V této kapitole jsou ukázány výsledky pro různé implementované algoritmy.

5.1 Porovnání přístupů ke tvorbě vstupního dekodéru

V této kapitole je porovnání výsledků pro dva přístupy ke tvorbě vstupních dekodérů popsané v kapitole 3.4.2. V tabulce 5.1 je ukázán odhad zdrojů a počet zabraných zdrojů po syntéze nástrojem ISE 11.3 pro standardní dekodér z kódu 1 z n na binární kód. V tabulce 5.2 je ukázán odhad zdrojů a počet zabraných zdrojů po syntéze nástrojem ISE 11.3 pro dekodér z kódu 1 z n na kódování stavů. Pro generování tohoto dekodéru bylo odpovídající mapování stavů vygenerováno náhodně. V tabulkách jsou hodnoty pro 64, 128, 256, 515, 1024 a 2048 vstupů.

Porovnáním tabulek je zřejmé, že druhý přístup je z pohledu zabraných zdrojů výhodnější než první, avšak je třeba míti na paměti, že velikost zabrané logiky čipu FPGA závisí na konkrétním mapování a může být vyšší než u prvního přístupu.

5.2 Porovnání přístupů k mapování tabulky přechodů na její implementaci

V tabulce 5.3 je srovnání obou přístupů popsaných v kapitole 2.4.2. První tři množiny pravidel jsou vytvořeny z pravidel používaných v reálném prostředí. Úspěšnost značí, v kolika procentech případů bylo dosaženo genetickým algoritmem stejného nebo lepšího výsledku než heuristikou. U genetického algoritmu je úspěšnost vypočítána na základě 20 běhů algoritmu. Pro všechny běhy a výpočty byly použity stejné parametry odpovídající výchozím

Tabulka 5.1: Zabrané zdroje na čipu a frekvence pro FPGA Virtex 5 LX155T pro standardní dekodér z kódu 1 z n na binární kód.

Vstupů [-]	Odhad LUT [-]	Odhad zpoždění [LUT]	Syntéza LUT [-]	Frekvence [MHz]
64	36	2	49	170
128	84	3	100	154
256	200	3	212	142
512	459	4	474	122
1024	1020	4	1032	96
2048	2233	4	2205	67

Tabulka 5.2: Zabrané zdroje na čipu a frekvence pro FPGA Virtex 5 LX155T pro dekodér z kódu 1 z n na kódování stavů.

Vstupů [-]	Odhad LUT [-]	Odhad zpoždění [LUT]	Syntéza LUT [-]	Frekvence [MHz]
64	25	2	37	156
128	51	3	68	153
256	98	3	130	141
512	197	4	256	122
1024	406	4	516	96
2048	789	4	1010	67

Tabulka 5.3: Porovnání výsledků mapování tabulky přechodů na její implementaci pro algoritmus heuristiky a genetického algoritmu. Výsledky jsou měřeny v celkovém počtu potřebných řádků paměti.

Množina RV	Stavů	Přechodů	Heuristika	GA	Generace	Úspěšnost [%]
1	21	5376	5376	5376	120	100
2	28	7168	7168	7168	713	30
3	174	44544	44544	44544	947	5
4	474	474	511	508	845	10
5	257	11889	12982	13587	954	0
6	475	16815	16885	17046	910	0

hodnotám uvedeným v tabulce 4.2. Další tři množiny jsou náhodně vytvořené. Z výsledků je vidět, že oba dva přístupy jsou schopné dosáhnou velice dobrých výsledků. Ve třech případech dosáhly nejlepšího možného mapování - počet zabraných řádků je roven počtu přechodů. U dalších výsledků není možné říct zda jsou nejlepší možné. Dále je z výsledků patrné, že v některých případech je lepší použití heuristiky a v jiných případech je lepší využití genetického algoritmu. Z tabulky je také vidět, že se vzrůstajícím počtem stavů a přechodů klesá v důsledku zvětšování stavového prostoru možných řešení úspěšnost navrženého genetického algoritmu. Výhodou heuristiky je to, že je schopná vždy vypočítat řešení. Výhodou i nevýhodou heuristiky je to, že produkuje pro stejný problém pokaždé stejné řešení. Výhodou genetického algoritmu je potenciální možnost nalezení lepšího řešení než je řešení vytvořené heuristikou.

5.3 Porovnání výhod použití sdílení tříd znaků oproti pouhému použití sdíleného dekodéru

Tato kapitola pojednává o výsledcích dosažených pro algoritmus vyhledávání regulárních výrazů se sdíleným dekodérem a pro algoritmus se sdíleným dekodérem a se sdílením tříd znaků. Pro porovnání jsou použity množiny pravidel programu L7 dekodér (vybrané) a programu Snort. V tabulce 5.4 je uveden počet stavů a přechodů pro jednotlivé množiny regulárních výrazů. Počet přechodů je uveden jak pro použití tříd znaků, tak bez tříd znaků. V tabulce 5.5 je uveden odhad spotřebovaných zdrojů FPGA Virtex 5 pro oba dva algoritmy rychlého vyhledávání regulárních výrazů. Z odhadu je vidět, že množství spotřebovaných zdrojů pro algoritmus se sdílením tříd znaků je mnohem menší. V tabulce 5.6 je uvedeno množství spotřebovaných zdrojů pro FPGA Virtex 5 LX155t po syntéze nástrojem ISE

Tabulka 5.4: Porovnání počtu stavů a přechodů NFA implementace používající sdílený dekodér (SD) a implementace používající sdílený dekodér a sdílení tříd znaků (SD + STZ).

Množina RV	Stavy [-]	SD Přechody [-]	SD + STZ Přechody [-]
l7	999	21599	1114
backdoor	4681	54130	5244
exploit	19812	4663117	20205
oracle	33555	8251348	33866
smtp	10110	1541826	11022
spyware-put	15180	274387	16566
voip	2117	307811	2283

Tabulka 5.5: Porovní odhadovaných zdrojů implementace používající sdílený dekodér (SD) a implementace používající sdílený dekodér a sdílení tříd znaků (SD + STZ)

Množina RV	SD LUT [-]	SD FF [-]	SD + STZ LUT [-]	SD + STZ FF[-]
l7	13759	836	1538	836
backdoor	22651	4043	4680	4043
exploit	1858354	19390	21471	19390
oracle	3281950	33274	34355	33274
smtp	617341	9909	10953	9909
spyware-put	112260	13189	13675	13189
voip	123058	1922	3128	1922

11.3. V tabulce 5.7 je uvedena dosažitelná frekvence a propustnost. Volné části tabulky značí případy, kdy syntézní nástroj nebyl schopen syntézu provést. Z výsledků je vidět, že optimalizátor syntézního nástroje je schopen optimalizovat (vytvořit třídy znaků) VHDL kód obsahující implementaci pouze se sdíleným dekodérem, ale za cenu značného prodloužení času syntézy a zvětšení rizika, že se syntéza nepovede.

Další zajímavým zjištěním je to, že syntézní nástroj je schopen do určité míry provádět i sbalení určitých sekvencí stavů do posuvného registru podobně jak to zavádí přístup 2.2.1. Počet vytvořených posuvných registrů pro jednotlivé množiny regulárních výrazů je uveden v tabulce 5.8. Na příkladu regulárních výrazů l7 dekodéru si můžeme ukázat, že při syntéze byly nalezeny všechny možné sekvence, které mohou být mapovány do posuvného registru.

Tabulka 5.6: Porovnání spotřebovaných zdrojů implementace používající sdílený dekodér (SD) a implementace používající sdílený dekodér a sdílení tříd znaků (SD + STZ) po syntéze nástrojem ISE 11.3 pro FPGA Virtex 5 LX155T

Množina RV	SD LUT [-]	SD FF [-]	SD + STZ LUT [-]	SD + STZ FF[-]
l7	1404	769	806	768
backdoor	3666	3606	3511	3606
exploit			18196	19097
oracle			32976	33238
smtp			9795	9760
spyware-put			9786	10159
voip			1786	1757

Tabulka 5.7: Porovnání dosažitelné frekvece a propustnosti pro implementace používající sdílený dekodér (SD) a implementace používající sdílený dekodér a sdílení tříd znaků (STZ) po syntéze nástrojem ISE 11.3 pro FPGA Virtex 5 LX155T

Množina RV	SD Frek. [Mhz]	SD Prop. [Gb/s]	STZ Frek. [Mhz]	STZ Prop. [Gb/s]
l7	148	1.184	351	2.808
backdoor	298	2.384	362	2.896
exploit			362	2.896
oracle			457	3.656
smtp			396	3.168
spyware-put			417	3.336
voip			358	2.864

Tabulka 5.8: Počet vytvořených posuvných registrů implementace používající sdílený dekodér a sdílení tříd znaků po syntéze nástrojem ISE 11.3 pro FPGA Virtex 5 LX155T

Množina RV	Posuvné registry [-]
l7	2
backdoor	0
exploit	37
oracle	11
smtp	0
spyware-put	0
voip	0

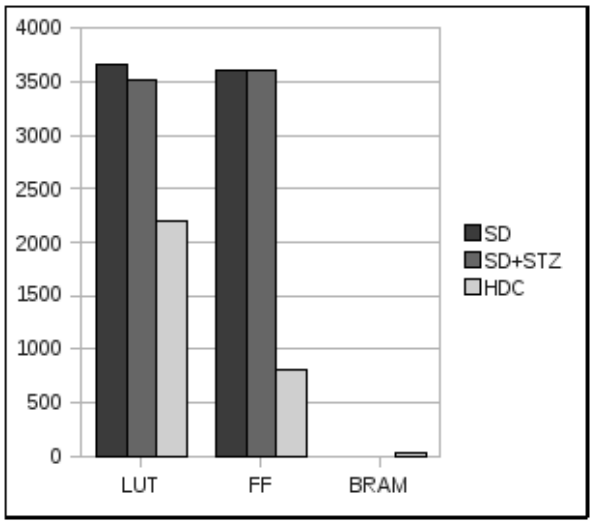
V této množině jsou to opakování 15 a 4 libovolných znaků, tj. celkem dva posuvné registry, což odpovídá hodnotě uvedené v tabulce 5.8.

5.4 Výsledky pro hledání deterministických částí

V tabulce 5.9 jsou uvedeny výsledky odhadu zdrojů pro implementaci jednotky pro vyhledávání zdrojů založenou na algoritmu hledání deterministických částí. Získané výsledky je možné porovnat s výsledky předcházející kapitoly (viz graf 5.1). Z porovnání výsledků je vidět, že algoritmus hledání deterministických částí dává lepší výsledky než obě dvě implementace založené pouze na nedeterministickém konečném automatu. V tabulce 5.10 je zobrazeno jaká část stavů byla umístěna do deterministické části a kolik jich zůstalo v nedeterministické části. Na obrázku 5.2 je zobrazen histogram počtu naráz aktivních stavů NFA pro množinu regulárních výrazů backdoor. Z histogramu je vidět, že v naprosté většině případů je najednou aktivních méně než 10 stavů. V jednom případě bylo najednou aktivních 122 stavů NFA.

Tabulka 5.9: Porovnání odhadu spotřebovaných zdrojů implementace používající algorimus hledání deterministických částí a skutečně spotřebovaných zdrojů po syntéze nástrojem ISE 11.3 pro FPGA Virtex 5 LX155T

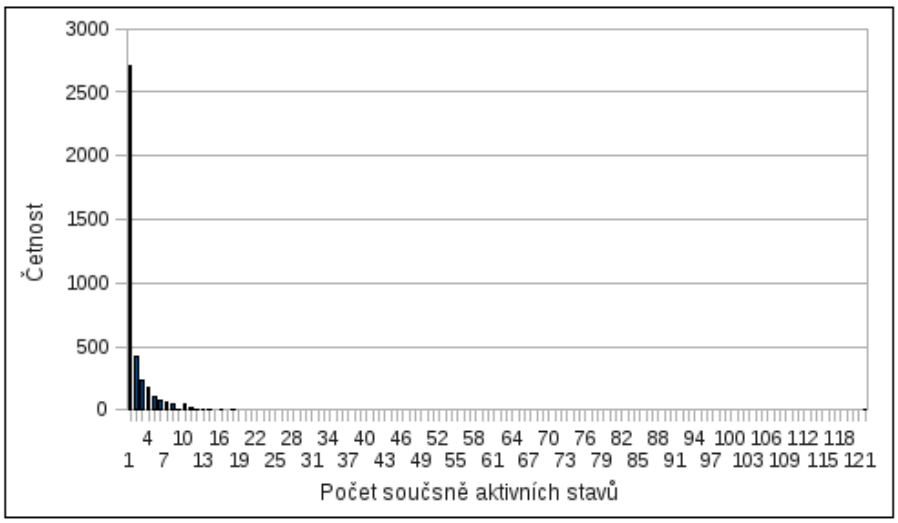
Množina RV	Odhad LUT	Odhad FF	Odhad BRAM	LUT	FF	BRAM
l7	1245	237	6	1165	234	6
backdoor	2464	821	27	2206	815	27



Obrázek 5.1: Porovnání počtu zabraných zdrojů pro množinu regulárních výrazů backdoor pro algoritmus vyhledávání vzorů se sdíleným dekodérem (SD), se sdíleným dekodérem a se sdílením tříd znaků (SD+STZ) a pro algoritmus hledání deterministických částí (HDC)

Tabulka 5.10: Počet stavů v deterministické a nedeterministické části.

Množina RV	Stavů celkem	DFA část	NFA část
l7	806	632	174
backdoor	3888	3250	638



Obrázek 5.2: Histogram počtu naráz aktivních stavů NFA.

Kapitola 6

Závěr

Seznámil jsem se s platformou COMBOv2 a technologií Virtex 5. Nastudoval jsem algoritmy pro rychlé vyhledávání vzorů, pro vybrané algoritmy jsem provedl návrh hardwarové implementace včetně ustanovení metodiky odhadu zabraných zdrojů. Navrhl jsem a implementoval softwarový systém pro generování jednotek pro vyhledávání vzorů. Navržený systém je snadno použitelná, rozšiřitelná a flexibilní.

Z dosažených výsledků je patrné, že optimalizace přináší významný efekt. I když je například optimalizátor syntézního nástroje ISE schopen dosáhnou pro pouhé sdílení dekodéru znaků výsledku podobného sdílení tříd znaků, je to vykoupeno značně delším během (řádově v desetinásobcích) a velkým rizikem neúspěchu. Dále je z výsledků vidět, že algoritmus hledání deterministických částí přináší snížení počtu spotřebovaných LUT a FF za cenu nutnosti použít paměti BRAM (případně externí paměť). Výsledky také potvrdily, že v nedeterministickém konečném automatu je většinou současně aktivních minimum stavů. Například pro množinu regulárnívh výrazů systému Snort backdoor, která má celkem 3888 stavů je v naprosté většině případů najednou aktivních méně než 10 stavů. Pouze v jednom případě bylo aktivních najednou 122 stavů a v 2699 případech byl najednou aktivní jen jeden stav.

Byly implementovány následující algoritmy pro rychlé vyhledávání vzorů:

- Algoritmus NFA implementace se sdíleným dekodérem znaků.
- Algoritmus NFA implementace se sdíleným dekodérem a se sdílením tříd znaků.
- Hybridní algoritmus hledání deterministických částí.

Další možná rozšíření softwarového systému:

- Podpora zpracování více znaků v jednom hodinovém cyklu, což přináší zvýšení dosažitelné propustnosti implementací.
- Další oblastí je podpora jiných způsobů zápisu regulárních výrazů než je používaný PCRE. Jedná se například o podporu zpracování kompletních pravidel IDS systému Snort a Bro.
- Dalším možným rozšířením je implementace dalších algoritmů rychlého vyhledávání vzorů.
- Zvýšení výkonnosti algoritmů přepsáním kritických částí do jazyka C.

- Použití tříd znaků pro algoritmy založené na DFA a pro algoritmy založené na NFA použít tuto optimalizaci v předřadném dekodéru.
- Rozšiřování a korekce implementace hybridního algoritmu pro vyhledávání vzorů - Hledání deterministických částí, jelikož je tento algoritmus stále vyvíjen.

Předpokládá se, že systém bude používán v projektu ANT pro implementace algoritmů rychlého vyhledávání výrazů vyvíjených na Ústavu počítačových systémů Fakulty informačních technologií Vysokého učení technického v Brně.

Literatura

[1] PCRE - Perl Compatible Regular Expressions [online]. `http://www.pcre.org/`, 29.10.2008 [cit. 5.1.2010].

[2] Becchi, M.; Crowley, P.: A hybrid finite automaton for practical deep packet inspection. In *CoNEXT '07: Proceedings of the 2007 ACM CoNEXT conference*, ACM, 2007, ISBN 978-1-59593-770-4, s. 1–12, doi:http://doi.acm.org/10.1145/1364654.1364656.

[3] Becchi, M.; Crowley, P.: Efficient regular expression evaluation: theory to practice. In *ANCS '08: Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ACM, 2008, ISBN 978-1-60558-346-4, s. 50–59, doi:http://doi.acm.org/10.1145/1477942.1477950.

[4] Bonesana, I.; Paolieri, M.; Santambrogio, M. D.: An adaptable FPGA-based system for regular expression matching. In *DATE '08: Proceedings of the conference on Design, automation and test in Europe*, ACM, 2008, ISBN 978-3-9810801-3-1, s. 1262–1267, doi:http://doi.acm.org/10.1145/1403375.1403681.

[5] Brodie, B. C.; Taylor, D. E.; Cytron, R. K.: A Scalable Architecture For High-Throughput Regular Expression Pattern Matching. *SIGARCH Computer Architecture News*, ročník 34, č. 2, 2006: s. 191–202, ISSN 0163-5964, doi:http://doi.acm.org/10.1145/1150019.1136500.

[6] Clark, C. R.; Schimmel, D. E.: Scalable Pattern Matching for High Speed Networks. In *FCCM '04: Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, IEEE Computer Society, 2004, ISBN 0-7695-2230-0, s. 249–257.

[7] Korenek, J.; Kosar, V.: Efficient Mapping of Nondeterministic Automaton to FPGA for Fast Regular Expression Matching. In *DDECS '10: Proceedings of the 2010 13th IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems*, IEEE Computer Society, 2010.

[8] Kumar, S.; Dharmapurikar, S.; Yu, F.; aj.: Algorithms to accelerate multiple regular expressions matching for deep packet inspection. In *SIGCOMM '06: Proceedings of the 2006 conference on Applications, technologies, architectures, and protocols for computer communications*, ACM, 2006, ISBN 1-59593-308-5, s. 339–350, doi:http://doi.acm.org/10.1145/1159913.1159952.

[9] Kumar, S.; Turner, J.; Williams, J.: Advanced algorithms for fast and scalable deep packet inspection. In *ANCS '06: Proceedings of the 2006 ACM/IEEE symposium on*

Architecture for networking and communications systems, ACM, 2006, ISBN 1-59593-580-0, s. 81–92, doi:http://doi.acm.org/10.1145/1185347.1185359.

[10] Liberouter: Our Hardware - Description of COMBO cards [online]. `http://www.liberouter.org/hardware.php?flag=2`, [cit. 5.1.2010].

[11] Lin, C.-H.; Hsiao, H.-S.: Hierarchical state machine architecture for regular expression pattern matching. In *GLSVLSI '09: Proceedings of the 19th ACM Great Lakes symposium on VLSI*, ACM, 2009, ISBN 978-1-60558-522-2, s. 133–136, doi:http://doi.acm.org/10.1145/1531542.1531576.

[12] Lin, C.-H.; Huang, C.-T.; Jiang, C.-P.; aj.: Optimization of pattern matching circuits for regular expression on FPGA. *IEEE Trans. Very Large Scale Integr. Syst.*, ročník 15, č. 12, 2007: s. 1303–1310, ISSN 1063-8210, doi:http://dx.doi.org/10.1109/TVLSI.2007.909801.

[13] Meduna, A.: *Automata and languages :theory and applications*. Springer, 2000, ISBN ISBN 1-85233-074-0.

[14] Rohrer, J.; Atasu, K.; van Lunteren, J.; aj.: Memory-efficient distribution of regular expressions for fast deep packet inspection. In *CODES+ISSS '09: Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, ACM, 2009, ISBN 978-1-60558-628-1, s. 147–154, doi:http://doi.acm.org/10.1145/1629435.1629456.

[15] Sidhu, R.; Prasanna, V. K.: Fast Regular Expression Matching Using FPGAs. In *FCCM '01: Proceedings of the the 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, IEEE Computer Society, 2001, ISBN 0-7695-2667-5, s. 227–238, doi:http://dx.doi.org/10.1109/FCCM.2001.22.

[16] Smith, R.; Estan, C.; Jha, S.; aj.: Deflating the big bang: fast and scalable deep packet inspection with extended finite automata. *SIGCOMM Comput. Commun. Rev.*, ročník 38, č. 4, 2008: s. 207–218, ISSN 0146-4833, doi:http://doi.acm.org/10.1145/1402946.1402983.

[17] Sourdis, I.; Bispo, J.; Cardoso, J. M. P.; aj.: Regular Expression Matching in Reconfigurable Hardware. *Journal of Signal Processing Systems*, ročník 51, č. 1, 2008: s. 99–121.

[18] Xilinx: *Virtex-5 FPGA User Guide*. 2009, `http://www.xilinx.com/support/documentation/user_guides/ug190.pdf`.

[19] Yang, Y.-H. E.; Jiang, W.; Prasanna, V. K.: Compact architecture for high-throughput regular expression matching on FPGA. In *ANCS '08: Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ACM, 2008, ISBN 978-1-60558-346-4, s. 30--39, doi:http://doi.acm.org/10.1145/1477942.1477948.

[20] Yu, F.; Chen, Z.; Diao, Y.; aj.: Fast and memory-efficient regular expression matching for deep packet inspection. In *ANCS '06: Proceedings of the 2006 ACM/IEEE symposium on Architecture for*

networking and communications systems, ACM, 2006, ISBN 1-59593-580-0, s. 93--102, doi:http://doi.acm.org/10.1145/1185347.1185360.

Dodatek A

Obsah CD

- Tento dokument ve formátu PDF (soubor dp.pdf).
- Tento dokument ve zdrojovém kódu LATEX (v adreáři doc_src).
- Zdrojové kódy (v adresáři src).
- Dokumentaci zdrojového kódu ve formátu html (v adresáři src/doc, vygenerováno nástrojem pydoc).
- Programová dokumentace (soubor prog_doc.pdf).

Dodatek B

Seznam zkratek

ASCII Běžný způsob kódování znaků v počítači.

ASIC Aplikačně specifický integrovaný obvod.

BRAM Bloková paměť typu RAM. Základní paměťový element FPGA.

CD^2FA Content Addressed Delayed Input DFA.

CPU Procesor.

DCAM Distribuovaná kontextově adresovaná paměť.

D^2FA Delayed Input DFA.

DFA Deterministický konečný automat.

DKA Deterministický konečný automat.

FF Klopný obvod.

GA Genetický algoritmus.

HDC Algoritmus vyhledávání vzorů: Hledání deterministických částí.

HDL Jazyk pro popis HW. V rámci této práce pod touto zkratkou rozumíme kód zapsaný v jazyce pro popis HW VHDL.

HFA Hybridní konečný automat.

HW Hardware.

IDS Systém pro detekci nežádoucího provozu.

k-NFA Rozšířený víceznakový konečný automat.

KA Konečný automat.

LUT Vyhledávací tabulka. Základní logický element FPGA, zpravidla 4 nebo 6 vstupá.

mDFA Algoritmus pro vyhledávání vzorů rozdělující jeden velký deterministický konečný automat na větší množství menších automatů.

NDKA Nedeterministický konečný automat.

NFA Nedeterministický konečný automat.

PCRE Regulární výraz kompatibilní s jazykem Perl.

RV Regulární výraz.

SD Algoritmus vyhledávání vzorů používající sdílený dekodér.

STZ Sdílení tříd znaků.

SD+STZ Algoritmus vyhledávání vzorů používající sdílený dekodér a sdílení tříd znaků.

SRL16 Posuvný registr délky 16 bitů. Logický element FPGA, nejčastěji implementovný pomocí LUT.

SRL32 Posuvný registr délky 32 bitů. Logický element FPGA od Virtex 5, nejčastěji implementovný pomocí LUT.

SW Software.

VLIW Procesor s velmi dlouhým instrukčním slovem.

XFA Extended Finite Automata.

Dodatek C

Manuál

Tato příloha pojednává jakým způsobem vytvořit implementovanými algoritmy HW reprezentaci jednotky v jazyce VHDL pro danou množinu regulárních výrazů. Generování je prováděno pomocí jednoduchých skriptů naimplementovaných v jazyce Python.

Pro použití je třeba mít nainstalovanou verzi jazyka Python 2.6. Jiné verze jazyka Python nejsou podporovány a systém na nich pravděpodobně nebude běžet správně. Pro použití systému se předpokládá operační systém Linux.

Vygenerované jednotky jsou zapsány v jazyce pro popis hardware VHDL. Syntéza byla zkoušena v nástroji Xilinx ISE 11.3.

C.1 Generování jednotky algoritmem se sdíleným dekodérem

Pro generování jednotky algoritmem se sdíleným dekodérem slouží skript gen_sd.py. Parametry tohoto skriptu jsou uvedeny v tabulce C.1 a jejich pozice v příkazové řádce je pevná.

Příklad použití: ./gen_sd.py ./jednotka_sd rules/l7/selected/selected.pcre

C.2 Generování jednotky algoritmem se sdíleným dekodérem a se sdílenými třídami znaků

Pro generování jednotky algoritmem se sdíleným dekodérem a se sdílenými třídami znaků slouží skript gen_sd_stz.py. Parametry tohoto skriptu jsou uvedeny v tabulce C.2 a jejich pozice v příkazové řádce je pevná.

Příklad použití: ./gen_sd_stz.py ./jednotka_sd_stz rules/l7/selected/selected.pcre

Tabulka C.1: Parametry a jejich pořadí pro skript gen_sd.py

Pořadí	Význam
1	Jméno adresáře kam bude uložena výstupní implementacce jednotky. Ve výsledném adresáři bude jeden VHDL soubor.
2	Jméno souboru s množinou regulárních výrazů.

Tabulka C.2: Parametry a jejich pořadí pro skript gen_sd_stz.py

Pořadí	Význam
1	Jméno adresáře kam bude uložena výstupní implementacce jednotky. Ve výsledném adresáři bude jeden VHDL soubor.
2	Jméno souboru s množinou regulárních výrazů.

Tabulka C.3: Parametry a jejich pořadí pro skript gen_hdc.py

Pořadí	Význam
1	Jméno adresáře kam bude uložena výstupní implementacce jednotky. Ve výsledném adresáři bude více VHDL souborů.
2	Jméno souboru s množinou regulárních výrazů.

C.3 Generování jednotky algoritmem hledání deterministických částí

Pro generování jednotky algoritmem se sdíleným dekodérem a se sdílenými třídami znaků slouží skript gen_hdc.py. Parametry tohoto skriptu jsou uvedeny v tabulce C.3 a jejich pozice v příkazové řádce je pevná.

Příklad použití: ./gen_hdc.py ./jednotka_hdc rules/l7/selected/selected.pcre